All registers and memory locations are 32 bits, the concept of *byte* does not apply except in the few special string-processing instructions. When characters are stored to make a string, they are packed four per memory location, with the first character of the string being in the least-significant 8 bits.

Negative numbers are represented in the two's complement format.

Floating point numbers are stored in the intel 32-bit floating format, whatever that is.

Bits are numbered from 0, the least significant, to 31 the most significant.
In numeric representations, bit 31 is the sign bit.

There are 16 regular registers, numbered from 0 to 15.
R0 is a scratch register, with slightly limited functionality
R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12 are general purpose registers
SP, the stack pointer, is encoded as register 13
FP, the frame pointer, is encoded as register 14
PC, the program counter, is encoded as register 15

The instruction format

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
Operation							I	Main Register				Index Register				Numeric Operand															

I is the Indirect bit. Two's complement, range -32768 to +32767

If bits 16-19 are all zero, i.e. "Index Register" indicates R0, then no index register is used when the instruction executes. Thus it is not possible to use R0 as an index register.

In the description of an instruction, the term *reg* refers to the register indicated by bits 20 to 23 (main register), and *operand* refers to the combination of indirect bit, index register, and numeric operand as illustrated on the next two pages.

If the term *value* appears in the description, it refers to the value of the operand, which is calculated as follows:

```
part1 = numeric operand;
part2 = 0;
if (index register ≠ 0)
      part2 = contents of indicated index register
total = part1 + part2;
if (indirect bit ≠ 0)
      value = contents of memory location [total];
else
      value = total;
```

If the sequence "*reg* ← *x*" appears, it means that the content of the main register is replaced by x.

If the sequence "*destination* ← *x*" appears, then the operand my consist of just an index register, in which case the content of the register is replaced by x, otherwise the indirect bit must be set, and the content of memory location [total] is replaced by x.

Assembly Examples:

```
   RET

0100101 0 0000 0000 0000000000000000
4A00000
```

Operation	= 37
Indirect bit	= 0
Main register	= 0
Index register	= 0
Numeric	= 0

```
   INC   R6

0000100 0 0110 0000 0000000000000000
0860000
```

Operation	= 4
Indirect bit	= 0
Main register	= 6
Index register	= 0
Numeric	= 0

```
   LOAD  R2, 36

0000001 0 0010 0000 0000000000100100
0220024
```

Operation	= 1
Indirect bit	= 0
Main register	= 2
Index register	= 0
Numeric	= 36

```
   ADD   R7, R3

0000110 0 0111 0011 0000000000000000
0C73000
```

Operation	= 6
Indirect bit	= 0
Main register	= 7
Index register	= 3
Numeric	= 0

```
   LOAD  R7, R3 + 12

0000001 0 0111 0011 0000000000001100
027300C
```

Operation	= 1
Indirect bit	= 0
Main register	= 7
Index register	= 3
Numeric	= 12

```
   ADD   R4, [R3]

0000110 1 0100 0011 0000000000000000
0D43000
```

Operation	= 6
Indirect bit	= 1
Main register	= 4
Index register	= 3
Numeric	= 0

```
   STORE R2, [1234]

0000011 1 0010 0000 0000010011010010
07204D2
```

Operation	= 3
Indirect bit	= 1
Main register	= 2
Index register	= 0
Numeric	= 1234

```
   STORE R2, [R5 - 375]

0000011 1 0010 0101 1111111010001001
0725FE89
```

Operation	= 3
Indirect bit	= 1
Main register	= 2
Index register	= 5
Numeric	= -375

Execution Examples, starting from these values already in memory:

location	*contents*
27100	592
27101	759
27102	43
27103	27105
27104	2
27105	682
27106	11
27107	22
27108	33

```
LOAD  R2, 5
```

The value stored in register 2 is now 5

```
LOAD  R3, R2+4
```

The value stored in register 3 is now 9

```
LOAD  R4, 27102
```

The value stored in register 4 is now 27102

```
LOAD  R5, [27100]
```

The value stored in register 5 is now 592

```
LOAD  R6, [R4]
```

The value stored in register 6 is now 43

```
ADD   R6, R2
```

The value stored in register 6 is now 48

```
STORE R6, [27101]
```

The content of memory location 27101 is changed from 759 to 48

```
INC   R6
```

The value stored in register 6 is now 49

```
STORE R6, [R4 - 2]
```

The content of memory location 27100 is changed from 592 to 49

```
LOAD  SP, 27108
```

The value stored in register 13 (stack pointer) is now 27108

```
PUSH  R2
```

The content of memory location 27107 is changed from 22 to 5
The value stored in register 13 (stack pointer) is now 27107

```
PUSH  [R4]
```

The content of memory location 27106 is changed from 11 to 43
The value stored in register 13 (stack pointer) is now 27106

```
POP   R4
```

The value stored in register 4 is now 43
The value stored in register 13 (stack pointer) is now 27107

```
STORE R6, 27101
```

Fails to execute, as the operand does not address memory.

opcode	*mnemonic*	*action*
0	HALT	the processor is halted, execution of instructions stops.
1	LOAD *reg*, *operand*	*reg* ← *value*
2	LOADH *reg*, *operand*	*reg* ← (*reg* ∧ FFFF) + (*value* « 16) the most significant 16 bits of the register are replaced
3	STORE *reg*, *operand*	*destination* ← *reg*
4	INC *operand*	*destination* ← *value* + 1
5	DEC *operand*	*destination* ← *value* - 1
6	ADD *reg*, *operand*	*reg* ← *reg* + *value*
7	SUB *reg*, *operand*	*reg* ← *reg* - *value*
8	MUL *reg*, *operand*	*reg* ← *reg* × *value*
9	DIV *reg*, *operand*	*reg* ← *reg* ÷ *value*
10	MOD *reg*, *operand*	*reg* ← *reg* modulo *value*
11	RSUB *reg*, *operand*	*reg* ← *value* - *reg*
12	RDIV *reg*, *operand*	*reg* ← *value* ÷ *reg*
13	RMOD *reg*, *operand*	*reg* ← *value* modulo *reg*
14	AND *reg*, *operand*	*reg* ← *reg* ∧ *value*
15	OR *reg*, *operand*	*reg* ← *reg* ∨ *value*
16	XOR *reg*, *operand*	*reg* ← *reg* ⊕ *value*
17	NOT *reg*, *operand*	*reg* ← ~ *value*
18	SHL *reg*, *operand*	*flagZ* ← 1 *if most sig. (value) bits of reg all* 0*, otherwise* 0 *reg* ← *reg* « *value,* zeros being inserted at the right
19	SHR *reg*, *operand*	*flagZ* ← 1 *if least sig. (value) bits of reg all* 0*, otherwise* 0 *reg* ← *reg* » *value,* zeros being inserted at the left
20	COMP *reg*, *operand*	*flagZ* ← 1 *if reg* = *value, otherwise* 0 *flagN* ← 1 *if reg* < *value, otherwise* 0
21	COMPZ *operand*	*flagZ* ← 1 *if value* = 0*, otherwise* 0 *flagN* ← 1 *if value* < 0*, otherwise* 0
22	TBIT *reg*, *operand*	*flagZ* ← value^{th} *bit of reg*
23	SBIT *reg*, *operand*	value^{th} *bit of reg* ← 1
24	CBIT *reg*, *operand*	value^{th} *bit of reg* ← 0

	25	`JUMP` *operand*	PC ← *value*
	26	`JZER` *reg*, *operand*	*if* (*reg* = 0) PC ← *value*
	27	`JPOS` *reg*, *operand*	*if* (*reg* ≥ 0) PC ← *value*
	28	`JNEG` *reg*, *operand*	*if* (*reg* < 0) PC ← *value*
	29	`JCOND`	Note that no main register is used with the `JCOND` instruction. Instead, its 4 bits are used to encode one of the seven condition tests shown here.
0		`JCOND EQL`, *operand*	*if* (*flagZ*) PC ← *value*
1		`JCOND NEQ`, *operand*	*if* (~ *flagZ*) PC ← *value*
2		`JCOND LSS`, *operand*	*if* (*flagN*) PC ← *value*
3		`JCOND LEQ`, *operand*	*if* (*flagZ* ∨ *flagN*) PC ← *value*
4		`JCOND GTR`, *operand*	*if* (~*flagZ* ∧ ~*flagN*) PC ← *value*
5		`JCOND GEQ`, *operand*	*if* (~*flagN*) PC ← *value*
	30	`GETFL` *reg, operand*	*reg* ← *flag*[*value*]
	31	`SETFL` *reg, operand*	*flag*[*value*] ← *reg*
	32	`GETSR` *reg, operand*	*reg* ← *specialregister*[*value*]
	33	`SETSR` *reg, operand*	*specialregister*[*value*] ← *reg*
	34	`PUSH` *operand*	SP ← SP - 1 *memory*[SP] ← *value*
	35	`POP` *operand*	*destination* ← *memory*[SP] SP ← SP + 1

	36	`CALL` *operand*	SP ← SP - 1
			memory[SP] ← PC
			PC ← *value*

	37	`RET`	PC ← *memory*[SP]
			SP ← SP + 1

	38	`LDCH` *reg*, *operand*	*value* is treated as a memory address. The *reg*th 8-bit byte (character) starting from that address in memory is loaded into *reg*. i.e., *reg* ← byte *(reg* `modulo` 4*)* of *memory*[*value* + *reg*÷4]
	39	`STCH` *reg*, *operand*	*value* is treated as a memory address. The *reg*th 8-bit byte (character) starting from that address is replaced by the value of register 0 without modifying the other 24 bits of that word. byte *(reg* `modulo` 4*)* of *memory*[*value* + *reg*÷4] ← R0
	40	`PERI`	Control peripheral activity: see separate documentation
	41	`FLAGSJ` *reg*, *operand*	*all flags* ← *reg* PC ← *value*

42	`PAUSE` *operand*	CPU idles for approximately *value* mS, unless interrupted
43	`BREAK`	Enter CPU single-stepping mode

44	`IRET`	PC ← *memory*[SP]
		all flags ← *memory*[SP+2]
		FP ← *memory*[SP+6]
		SP ← *memory*[SP+7]
		R12 ← *memory*[SP+8]
		R11 ← *memory*[SP+9]
		R10 ← *memory*[SP+10]
		R9 ← *memory*[SP+11]
		R8 ← *memory*[SP+12]
		R7 ← *memory*[SP+13]
		R6 ← *memory*[SP+14]
		R5 ← *memory*[SP+15]
		R4 ← *memory*[SP+16]
		R3 ← *memory*[SP+17]
		R2 ← *memory*[SP+18]
		R1 ← *memory*[SP+19]
		R0 ← *memory*[SP+20]
		SP ← SP + 21

45	`SYSCALL` *reg*, *code*	*memory*[SP-1] ← R0 *memory*[SP-2] ← R1 *memory*[SP-3] ← R2 *memory*[SP-4] ← R3 *memory*[SP-5] ← R4 *memory*[SP-6] ← R5 *memory*[SP-7] ← R6 *memory*[SP-8] ← R7 *memory*[SP-9] ← R8 *memory*[SP-10] ← R9 *memory*[SP-11] ← R10 *memory*[SP-12] ← R11 *memory*[SP-13] ← R12 *memory*[SP-14] ← SP *memory*[SP-15] ← FP *memory*[SP-16] ← *main register number* *memory*[SP-17] ← *code* *memory*[SP-18] ← *0* *memory*[SP-29] ← *all flags* *memory*[SP-20] ← 38 *memory*[SP-21] ← PC SP ← SP - 21 PC ← *memory*[*specialregister*[CGBR] + *code*] *flagSys* ← 1
46	`ATAS` *reg*, *operand*	*reg* ← *value ; destination* ← 1 *performed indivisibly, ignoring interrupts*

47	`PHLOAD` *reg*, *operand*	*reg* ← *physicalmemory*[*value*]
48	`PHSTORE` *reg*, *operand*	*physicalmemory*[*value*] ← *reg*
49	`VTRAN` *reg*, *operand*	*reg* ← physical address for virtual address *value*
50	`MOVE` *reg*, *reg2*	while R0 > 0 repeat { *memory*[*reg2*] ← *memory*[reg] *reg2* ← *reg2* + 1 *reg* ← *reg* + 1 R0 ← R0 - 1 }
51	`SIGN` *reg*, *operand*	*reg* ← −1 *if value* < 0, 0 *if* 0, 1 *if* > 0
52	`FADD` *reg*, *operand*	*floating point: reg* ← *reg* + *value*
53	`FSUB` *reg*, *operand*	*floating point: reg* ← *reg* - *value*
54	`FMUL` *reg*, *operand*	*floating point: reg* ← *reg* × *value*
55	`FDIV` *reg*, *operand*	*floating point: reg* ← *reg* ÷ *value*
56	`FCOMP` *reg*, *operand*	*floating point:* *flagZ* ← 1 *if reg* = *value, otherwise* 0 *flagN* ← 1 *if reg* < *value, otherwise* 0
57	`FCOMPZ` *operand*	*floating point:* *flagZ* ← 1 *if operand* = 0*, otherwise* 0 *flagN* ← 1 *if operand* < 0*, otherwise* 0
58	`FIX` *reg*, *operand*	*reg* ← (int)*value*, *value* interpreted as floating point
59	`FRND` *reg*, *operand*	*reg* ← (float)(closest int to *value*), *both floating point*
60	`FLOAT` *reg*, *operand*	*reg* ← (float)*value*, *value* interpreted as an integer
61	`FLOG` *reg*, *operand*	*floating point:* *reg* ← *natural log(reg), if value* = 0 *reg* ← *log base value(reg), otherwise*
62	`FEXP` *reg*, *operand*	*floating point:* *reg* ← *e to power(reg), if value* = 0 *reg* ←*value to power(reg), otherwise*
63	`FSQRT` *reg*, *operand*	*floating point: reg* ← *square root of value*
64	`FSIN` *reg*, *operand*	*floating point: reg* ← *sine of value*
65	`FCOS` *reg*, *operand*	*floating point: reg* ← *cosine of value*
66	`FATAN` *reg*, *operand*	*floating point: reg* ← *arc-tangent of value*

67	`FABS` *reg*, *operand*	*floating point: reg ← value if value >= 0, - value otherwise*
68	`FLOOR` *reg*, *operand*	*floating point: reg ← value rounded towards - ∞*
69	`FSIGN` *reg*, *operand*	*floating point: reg ← integers* −1 *if value* < 0*,* 0 *if* 0*,* 1 *if* > 0
70	`FFO` *reg*, *operand*	*reg* ← number of bits to right of leftmost 1 in *value* if *value* = 0: *reg* ← -1*, flagZ* ← 1*, flagN* ← 1
71	`FLZ` *reg*, *operand*	*reg* ← number of bits to right of rightmost 0 in *value* if *value* = -1: *reg* ← -1*, flagZ* ← 1*, flagN* ← 1
72	`RAND` *reg*	*reg* ← random positive number
73	`TRACE` *reg, operand*	display PC*, reg,* and *value* on console
74	`TYPE` *operand*	send single character *value* to controlling teletype
75	`INCH` *operand*	*destination* ← one character code from controlling keyboard or -1 if none available
76	`ANDN` *reg*, *operand*	*reg* ← *reg* ∧ ~ *value*
77	`ORN` *reg*, *operand*	*reg* ← *reg* ∨ ~ *value*
78	`NEG` *reg*, *operand*	*reg* ← - *value*
79	`FNEG` *reg*, *operand*	*reg* ← - *value, value* interpreted as floating point
80	`ROTL` *reg*, *operand*	*reg* is shifted *value* bits left, with the bits lost at the left being reinserted at the right.
81	`ROTR` *reg*, *operand*	*reg* is shifted *value* bits right, with the bits lost at the right being reinserted at the left.
82	`ASR` *reg*, *operand*	*flagZ* ← 1 *if least sig. (value) bits of reg all* 0*, otherwise* 0 *reg* ← *reg* » *value,* the sign bit being duplicated at the left
83	`EXBR` *reg*, *operand*	R0 ← bit range described by *reg* from *value,* with the most significant bit of the range giving the sign.
84	`EXBRV` *reg*, *operand*	R0 ← bit range described by *reg* of *value,* with the most significant bit of the range giving the sign.
85	`DPBR` *reg*, *operand*	bit range described by *reg* from *value* ← R0.
86	`DPBRV` *reg*, *operand*	bit range described by *reg* of *value* ← R0.
87	`ADJS` *reg*, *operand*	the bit range selector in *reg* is advanced by *value* positions, taking into account the range size and the requirement for ranges not to span two words. *value* may be negative.

88	`UEXBR` *reg*, *operand*	R0 ← bit range described by *reg* from *value,* unsigned.
89	`UEXBRV` *reg*, *operand*	R0 ← bit range described by *reg* of *value,* unsigned.
90	`UCOMP` *reg*, *operand*	*flagZ* ← 1 *if reg* = *value, otherwise* 0 *flagN* ← 1 *if reg* < *value, otherwise* 0, an unsigned comparison
91	`UMUL` *reg*, *operand*	*reg* ← *reg* × *value*, unsigned
92	`UDIV` *reg*, *operand*	*reg* ← *reg* ÷ *value*, unsigned
93	`UMOD` *reg*, *operand*	*reg* ← *reg* `modulo` *value*, unsigned
94	`CLRPP` *operand*	page containing physical address *value* all set to zero

95	`ZERO` *reg*, *reg2*	(operand may only be a register, *reg2*)
		while *reg* > 0 repeat
		{ *memory*[*reg2*] ← 0
		  reg2 ← *reg2* + 1
		  reg ← *reg* - 1 }

96	`LBITF` *reg*, *operand*	*value* is treated as a memory address. The *reg*th bit starting from that address in memory is loaded into *reg*. i.e., *reg* ← bit *(reg* `modulo` 32*)* of *memory*[*value* + *reg*÷32]
97	`LBITO` *reg*, *operand*	The *reg*th bit of *value* is loaded into *reg*.
98	`SBITF` *reg*, *operand*	*value* is treated as a memory address. The *reg*th bit starting from that address is replaced by the value of register 0 without modifying the other 31 bits of that word. i.e., bit *(reg* `modulo` 32*)* of *memory*[*value* + *reg*÷32] ← R0
99	`SBITO` *reg*, *operand*	*operand* ← *value* with its *reg*th bit set to bit 0 of R0.
100	`PMEMR` *reg*, *operand*	writes up to *reg* words into memory starting at *destination*, *reg* must be an even number, each pair of words written represents an area of memory starting with its first address and ending with its last, the first pair gives the area that the program was loaded into, subsequent pairs describe areas that exist, in order. *reg* ← number of words needed to cover all memory

101	`FFNZ` *reg*, *operand*	while *reg* <= *operand* repeat
		{ if *memory*[*reg*] ≠ 0
		    return
		  reg ← *reg* + 1 }
		reg ← −1

102	`NOP`	nothing happens.
103	`SEXT` *reg*, *operand*	bit number *operand* of *reg* is copied into every bit to the left

		of it (sign extension). The least significant bit is bit 0.
104	`INTR` *reg*, *operand*	interrupt whose number is given by the operand is raised. The value of *reg* provides the "interrupt detail" that is pushed onto the system stack along with the other state information.
105	`MPUSH` *reg*, *operand*	Multiple registers, determined by the operand value are pushed onto the stack, the smallest numbered registers first. Each bit in the operand represents one register, the least significant being R0. An operand value of 70 (1000110_2) would push R1 then R2 then R6.
106	`MPOP` *reg*, *operand*	The opposite of `MPUSH`. The indicated registers are popped from the stack, smallest numbered last.
107	`FGOOD` *operand*	*flagZ* ← 1 *if operand is proper float number, otherwise* 0
127	`NALT`	same as HALT, used to trap accidental execution of data.

Special Registers

There are fourteen special registers, as follows

`FLAGS`	A single word containing all of the one-bit flags
`PDBR`	Page Directory Base Register
`INTVEC`	The address of the interrupt vector
`CGBR`	Call Gate Base Register
`CGLEN`	Number of call gates
`DEBUG`	If the value of PC ever = 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.
`WATCH`	Interrupt if address ever gets written to.
`EXITCODE`	Exit code to be delivered to the host unix system.
`IPL`	Interrupt processing level.
`EMGRET`	Emergency return address, described alongside interrupts below.

The assembler understands the names of these registers (put a `$` sign in front), they stand for the numbers 0 to 13 in instruction operands. The BCPL compiler also understands them if you put `SR$` at the front.

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.
`SYS`	Set when CPU is in system mode, Zero when in user mode.
`EM`	An emergency interrupt procedure is in progress, described under interrupts below.
`INT`	Interrupts will be processed, initially 0.
`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 four, `SYS`, `EM`, `INT`, and `VM`, may only be modified when the CPU is in system mode.

At start-up, R=1, SYS=1, EM=0, INT=0, VM=0.

The assembler understands the names of these flags (put a $ sign in front), they stand for the numbers 5 to 11 in instruction operands. The BCPL compiler also understands them in two ways: put FLAG$ in front of the name and you get the interrupt number, put FLAG$MASK in front of the same and you get the corresponding bit-mask, a bitwise and of the value from the sr$flags special register and the flag$maskem constant is non-zero if the EM flag is on.

There are two instructions that directly access the special registers:
GETFL loads the value of a single flag into a register
SETFL sets a single flag equal to a register value (0 for off, non-0 for on)
The COMP and COMPZ instructions set or clear both Z and N, depending on the result.
The JCOND instruction jumps if the flags have a particular combination of values.

All the flag values may be read at once, using the GETSR instruction on the $FLAGS special register. The least significant five bits of the flags register are the IPL special register, it has no separate existence. The flags occupy the next seven bits of the value, in the order shown above. R bit 5 and INT is bit 11 (equivalent value 2048).

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 that represent a fatal problem (such as a user mode program attempting a privileged operation), there are interrupts that represent fixable problems (such as a page fault, described under virtual memory below), and there are interrupts that represent some useful notification (such as keyboard input ready, or countdown timer reached zero). If interrupts are being processed (that is, the INT flag is 1, and the INTVEC special 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 (INT flag is 0 or INTVEC is 0), then all interrupts except from TIMER and KEYBD (which are just ignored) will stop a running program.

If interrupts are being accepted (INT = 1) 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.

If interrupts are being accepted (INT = 1) and a particular interrupt arises, but the interrupt vector contains a zero at its position, a second interrupt, INTRFAULT, is signalled. This may also be trapped, and in this case there is a chance for your program to survive.

If an INTRFAULT interrupt occurs and the interrupt vector contains a zero at that position, the program will just stop.

One way to stop all interrupts from ever having any effect whatsoever is to temporarily set the special register IPL (interrupt processing level) to INTRFAULT, which is the highest priority interrupt. The first action in processing any interrupt is to make sure it is higher than IPL, and completely ignore it if it isn't.

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 if something goes wrong while an interrupt is still being processed, it can be hard to handle. There are a few safety mechanisms described at the end of this section

There are 21 interrupts defined, each with a name known to the assembler and the BCPL compiler, where their names are all prefixed with INT$. An interrupt vector is really an array, and must be at least 18 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, or in BCPL use an ireturn statement at the end of the handler and anywhere else it can exit.

The defined interrupts are:

NONE	= 0:	(not a real interrupt code)
HALT	= 1:	HALT instruction executed while in user mode
TIMER	= 2:	Countdown timer reached zero
KEYBD	= 3:	at least one keyboard character typed and ready
PRIVOP	= 4:	Privileged operation attempted by user mode program
PAGEPRIV	= 5:	User mode access to system mode page
DIVZERO	= 6:	Division by zero
MEMORY	= 7:	Physical memory access failed
UNIMPOP	= 8:	Unimplemented operation code (i.e. instruction opcode wrong)
UNWROP	= 9:	Unwritable instruction operand (e.g. INC 72)

BADOP = 10: Unsuitable operand for instruction.

BADCALL = 11: Bad SYSCALL index (i.e. < 0 or >= CGLEN)

PAGEFAULT = 12: Page fault

PAGEFAULT2 = 13: Page fault occurred during processing of an earlier page fault

WATCH = 14: Write to watch-point trap.

DEBUG = 15: PC=$DEBUG trap

SYSSTKFL = 16: Failure to process interrupt because the system stack is full.

INTRFAULT = 17: Failure to process interrupt for some other reason.

FLTNEG = 18: A floating point operation received an improper negative operand.

USRINT1 = 19: deliberately caused by an INTR R*x*, 19 instruction.

USRINT2 = 20: deliberately caused by an INTR R*x*, 20 instruction.

USRINT3 = 21: deliberately caused by an INTR R*x*, 21 instruction.

RTERROR = 22: deliberately caused by an INTR R*x*, 22 instruction to signal a run-time error. The emulator is able to produce tailor-made error messages based on the contents of the registers.

These values are the positions in the interrupt vector where the handler function’s address should be stored.

Example: How to set up an interrupt handler that automatically prints a dot whenever a keyboard key is pressed, and a star whenever another 5000 instructions have been executed...

```
     LOAD     R1, TIMHANDLER
     STORE    R1, [IVEC + INT$TIMER]
     LOAD     R1, KBHANDLER
     STORE    R1, [IVEC + INT$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
```

Actions Automatically Performed when an Interrupt Occurs, if INT flag is 1.

If an interrupt is already being processed when another interrupt occurs, and the code number (1 to 17) for this new interrupt is less than the code number for the interrupt already being processed (the value of the IPL special register), then the new interrupt is ignored. There is a special mechanism to ensure that when a KEYBD or TIMER interrupt is ignored, it will be reasserted when the IPL gets low enough to give it priority. Otherwise:

```
oldflags  = FLAGS register
flag SYS turned on.  (i.e. now using system SP and system stack)
special register IPL = code number for the interrupt.
PUSH R0
PUSH R1
...
...
PUSH R11
PUSH R12
PUSH SP
PUSH FP
PUSH  additional interrupt information if available
PUSH  interrupt-causing address
PUSH  interrupt code (i.e. position in interrupt vector)
PUSH  oldflags
PUSH  38
PUSH  PC
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 INT$ value for the interrupt.

For the following interrupts:

PAGEFAULT, PAGEFAULT2, PAGEPRIV,

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

For this interrupt:

MEMORY,

the second parameter is the address that caused the problem.

For this interrupt:

WATCH,

the second parameter is the address that was about to be written to.

For the following interrupts:

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

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 these interrupts:

INTRFAULT, SYSSTKFL,

which are 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.

For these interrupts:

USRINT1, USRINT2, USRINT3,

which are only caused deliberately by a program, the second parameter is whatever the cause, an INTR instruction provided through its main register.

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, but it is normal for all processes to share a single system stack.

WHEN THINGS GO WRONG WITH AN INTERRUPT.

Except for one case, once an interrupt handler has been successfully started everything behaves as normal. This section is mostly about when something goes wrong while the automatic actions described above are being carried out.

The exception is for a page fault. Normally only higher priority interrupts can interrupt the handling of another interrupt. Others, including a recurrence of the same interrupt are ignored. But page faults are slightly different. Whenever a page fault occurs, it must be handled or nothing will be able to work. To allow this, the mechanism for page faults is different. If a new page fault occurs during the handling of an earlier page fault, it is converted to a PAGEFAULT2, which has a slightly higher priority. But there is nothing beyond this, just this one second chance.

During the automatic actions, there are only a small number of things that can go wrong. It might be impossible to push those 22 values onto the system stack. Perhaps it is full or virtual memory is set up incorrectly. It is impossible to handle this as another higher priority interrupt because all interrupt processing requires the current state to be pushed onto the system stack. The emergency procedure to make this survivable has two steps: First, if there is a non-zero entry for SYSSTKFL in the interrupt vector, it will be jumped to, not called in the normal way. The EMGRET (emergency return) special register is set to the address of the next instruction after the jump, and the EM flag (emergency under way) is turned on. If the handler is able to fix the problem, then it should clear the EM flag and retrieve the address stored in EMGRET. EMGRET should then be zeroed before jumping to the address it used to contain. A normal IRET instruction would result in processing of the failed interrupt being terminated as though it has been successful. If this SYSSTKFL mechanism can not be performed, then exactly the same thing is tried but for INTRFAULT instead. If that fails, then a running program will be stopped and the emulator will enter single step mode for debugging.

Input and Output Operations

All interactions with any hardware outside of the CPU are controlled by the PERI instruction. The PERI instruction is accessible through the BCPL library function devctl. These operations produce a positive or zero result when the operation is successful and a negative error code when it is not. The error codes listed for each operation are available to programmers by prefixing err$ in a BCPL program or just $ in assembly language to the name, so the MEMORY error code would be err$memory or $MEMORY.

There are four general groups of IO operations supported:

Disc Operations: These allow direct access to the emulated disc drives, permitting whole blocks (128 words, which is the same size 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:        .DATA  $DISCCHECK
        .DATA   1
```

This style requires fewer instructions, but is slightly less flexible.
`PERI` is a privileged operation, and can not be executed in user mode.

If the operation code is not recognised, nothing happens except that the error code `-1` (`ERR_BAD_CODE`) is stored in the register.

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, READPARAMS`:	memory access problem reading the control structure.
`-3, DEVNUMBER`:	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 4 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: (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, READPARAMS`:	memory access problem reading the control structure.
`-3, DEVNUMBER`:	indicated disc not available
`-4, POSITION`:	attempt to read a block number < 0 or >= size.
`-5, MEMORY`:	memory access problem reading the data

Successful result (returned in register):

number of blocks transferred from disc to memory.

DISCWRITE

Requires a 4 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: (memory address) the address of the data to be written.
make sure that there are at least 128 words of memory set aside there.

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

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

DISCCLEAR

Completely erases a disc, setting every byte to zero.

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

Error codes same as for `DISCREAD` plus
`-2, READPARAMS`: memory access problem reading the control struction
`-6, DEVFAILED`: 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, READPARAMS`: memory access problem reading the control structure.
`-3, DEVNUMBER`: 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 word control structure, as follows
0: the value `TAPEREWIND`
1: the tape unit number

Error codes (returned in register):
`-2, READPARAMS`: memory access problem reading the control structure.
`-3, DEVNUMBER`: tape unit not available.

Repositions the tape to its beginning.

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, READPARAMS`: memory access problem reading the control structure.
`-3, DEVNUMBER`: tape unit not available.
`-5, MEMORY`: memory access problem reading the filename string
`-7, NOTFOUND`: the file is not accessible.
`-8, BADPARAM`: 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, READPARAMS`: memory access problem reading the control structure.
`-3, DEVNUMBER`: tape unit not available.
`-7, NOTFOUND`: 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 word control structure, as follows
0: the value `TAPEUNLOAD`
1: the tape unit number

Error codes (returned in register):
`-2, READPARAMS`: memory access problem reading the control structure.
`-3, DEVNUMBER`: tape unit not available.
`-7, NOTFOUND`: 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, READPARAMS`: memory access problem reading the control structure.
`-3, DEVNUMBER`: tape unit not available
`-5, 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, DEVFAILED: real failure to write all the data

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

TAPELOADFILE

Requires a 3 word control structure, as follows
0: the value TAPELOADFILE
1: a string, the name of the real unix file to be read
2: an address, the beginning of an area of free memory big enough to hold the whole file.

The entire file is read into memory in a single operation.

Error codes (returned in register):
-2, READPARAMS: memory access problem reading the control structure.
-5, MEMORY: memory access problem reading the filename string
-7, NOTFOUND: the file is not accessible.

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

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

```
LOAD    R1, control
LOAD    R2, $TAPELOAD
STORE   R2, [R1]
LOAD    R2, 1                  // unit number
STORE   R2, [R1+1]
LOAD    R2, filename
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 essential operations: read a bunch of characters from the keyboard and write a bunch of characters to the screen. The read function is compatible with interrupt-driven user input: when a program is running (not just single stepping) and interrupts are enabled, every time a keyboard key is pressed its ASCII code is added to the end of the hardware keyboard buffer and a `$KEYBD` interrupt is signalled. The `$TERMINC` operation takes characters from the beginning of the hardware keyboard buffer.

Character codes are available as soon 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.

TERMIN

Requires 3 word control structure, as follows
0: the value `TERMIN`
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 / 4 + 1 words of space there.

Error codes (returned in register):
`-2, READPARAMS`: memory access problem reading the control structure.
`-5, 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`.

TERMOUT

Requires 3 word control structure, as follows

0: the value `$TERMOUT`

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 `$TERMIN`

Successful result (returned in register):

the number of characters actually printed

Notes:

The characters to be printed should be in the form of a proper string (packed four per word) starting at the given memory location. The string does not *need* to be zero-terminated.

If the number of characters is specified to be zero, the string will be assumed to be zero-terminated, and an unlimited number of characters will be printed.

If the number of characters 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.

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 a 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, READPARAMS`: memory access reading or writing the control structure.
`-3, DEVNUMBER`: unit < 1 or > 2, or when closing a device, unit not in use.
`-5, MEMORY`: error while accessing the two-word IP address..
`-9, INUSE`: when starting a device, unit already in use.

NETSEND

Send a packet of data.
Requires 4 word control structure, as follows
0: the value `NETSEND`
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, READPARAMS`: memory access reading or writing the control structure.
`-3, DEVNUMBER`: unit < 1 or > 2, or when closing a device, unit not in use.
`-5, MEMORY`: error accessing IP address or data to be sent.
`-6, DEVFAILED`: the underlying unix call used to simulate IP transmission failed.
`-8, BADPARAM`: number of bytes < 0 or > 1024

NETRECV

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

TIME OPERATIONS

SECONDS

Requires 1 word control structure, as follows
0: the value SECONDS

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

Successful result (returned in register):
The number of seconds elapsed since midnight (0000 hours) on 1st January 2000.

USECONDS

Requires 3 word control structure, as follows
0: the value USECONDS
1: a memory address, there must be at least two free words starting here.

Error codes:
-2, READPARAMS: memory access reading or writing the control structure.
-5, MEMORY: error writing the time data to memory.

Successful result (returned in memory whose address was given):
Position 0: the number of seconds elapsed since midnight (0000 hours) on 1st January 2000.
Position 1: the number of microseconds elapsed since the beginning of that second.

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 or $USECONDS
If the value here is -1, the current date and time will be used.
2: a memory address, there must be at least seven free words starting here.

Error codes:
-2, READPARAMS: memory access reading or writing the control structure.
-5, MEMORY: error writing the time data to memory.

Successful result (returned in memory whose address was given):
Position 0: the year.
Position 1: the month.
Position 2: the day of the month.
Position 3: the day of the week, 0 means Sunday.
Position 4: the hour.
Position 5: the minute.
Position 6: the second.

This operation is something of a cheat. No real computer would have hardware for this task, but converting seconds to a date is very difficult. It would be easy, but you'd have to know the exact dates for daylight saving time for any year past or future, and those dates change sometimes.

Miscellaneous Operations

FLOATFORMAT

Requires 4 word control structure, as follows
0: the value `FLOATFORMAT`
1: a floating point number
2: a format string
3: a destination

Error codes:
`-2, ERR_READPARAMS`: memory access reading or writing the control structure.
`-5, MEMORY`: error accessing either of the strings.

The format string must contain a format that could be used exactly by C's `printf` function to print a floating point number. the destination must be the address of an area of free memory large enough to contain the string that `printf` would have printed. This is a bit of a cheat, no real computer would have hardware for such a purpose, but again it is very difficult to do properly.

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 means that there would be 4096 memory locations in a page, and that would mean that a page table could hold the addresses of 4096 pages instead of 1024, so we would not need so many of them.

In the emulator a page of memory consists of 2048 32-bit locations requiring only an 11 bit offset. That means that a page table can hold the addresses of 2048 pages, so 11 bits are required for page numbers. That leaves only 10 bits for the page table number, meaning that page directories only fill half a 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 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. The BCPL compiler has constants for referring to these bits: `page$validbitnum` and `page$systembitnum` (0 and 1) and `page$validmask` and `page$systemmask` (1 and 1). It also has constants that make the layout of page directory and table entries transparent: `page$size` (2048), `page$tablenumbits` (10), `page$pageintablebits` (11), `page$pagenumbits` (21), `page$offsetbits` (11), `page$tablenummask` (binary ten ones followed by 22 zeros), `page$pageintablemask` (binary ten zeros then eleven ones then eleven zeros), `page$pagenummask` (binary twenty-one ones followed by eleven zeros), and `page$offsetmask` (binary twenty-one zeros followed by eleven ones).

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 Valid bit is 0, 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 (using C/C++ syntax for expressions)

```
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 virtual address and will be translated.
    let DIR be (A >> 22) & 0x3FF           // most significant 10 bits
    let PG be (A >> 11) & 0x7FF            // next 11 bits
    let OFFS be A & 0x7FF                  // least significant 11 bits
    let POS be DIR + contents of PDBR register
    read PTADDR from physical memory address POS
    if PTADDR is Zero:
            PAGEFAULT, translation abandoned
    PTADDR &= 0xFFFFF800                   // zero out least significant 11 bits
    read PGADDR from physical memory address (PTADDR + PG)
    let V be PGADDR & 1                    // least significant bit
    if V 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, INT or EM flag, a PRIVOP interrupt will be triggered and the operation will not be performed.