The BLITZ Emulator

Harry H. Porter III, Ph.D.
Computer Science Department
Portland State University

Using The BLITZ Emulator

The BLITZ emulator is a program written in "C" which emulates the BLITZ architecture. In other words, the emulator is a virtual machine which simulates in software the behavior of a BLITZ machine. This program is named "blitz" and is run on a computer known as the "host" computer. When running under Unix, for example, you may start the emulator by typing "blitz" at the Unix prompt.

The emulator begins by reading in a BLITZ program and loading it into memory. Normally, the BLITZ executable file is called "a.out" but it can be given another name. The emulator begins by reading data from "a.out" and loading it into its internal memory. In effect, the emulator begins by initializing the main memory of the BLITZ machine, using the bytes in the "a.out" file.

As an example, assume there is a file called "test.s" containing a BLITZ assembly code program. The following sequence can be used to assemble, link, and run this program. In this document, "%" is the Unix prompt. We show user input as underlined, boldface.

```
% asm test.s
% lddd test.o
% blitz
```

The program "asm" is the BLITZ assembler. It takes as input an assembly language program and produces an object file called, in this case, "test.o".

The second program (called "lddd") is the BLITZ linker. It takes as input one or more object files and produces an executable file called "a.out". The executable file may be renamed with the "-o" command line option.

The program called "blitz" is the emulator. It loads an executable file into the main memory of the emulated BLITZ machine. By default, the emulator reads from a file called "a.out", but another file may be named on the command line.

The BLITZ emulator is command oriented. It accepts one command at a time and executes each command before prompting for the next command.

The emulator is meant to be run interactively, with "stdin" and "stdout" connected to an interactive user interface. The BLITZ emulator uses the ">" character as a prompt. You type in commands after this prompt and the result of each will be displayed.

One command is called "go"; this command begins executing BLITZ machine instructions. Other commands allow you to do things like:

```
(1)  Look at (and change) the BLITZ registers
(2)  Look at (and change) the contents of the BLITZ memory
(3)  View the state of the BLITZ machine
(4)  Execute a single instruction at a time
(5)  Dis-assemble instructions from memory
(6)  Manipulate the I/O devices (the serial I/O and the disk)
(7)  Quit the emulator
```

The "quit" Command

The "quit" command (which may be abbreviated as "q") will terminate the BLITZ emulator.

Before terminating, the emulator will print some execution statistics, reflecting all activity since the emulator began (or since the last "reset" command).

```
> quit
Number of Disk Reads    = 0
Number of Disk Writes   = 0
Instructions Executed   = 18560
Time Spent Sleeping     = 0
    Total Elapsed Time  = 18560
%
```

(The final "%" symbolizes the host / Unix prompt.)

The "go" Command

The "go" command (which may be abbreviated as "g") is used to start execution of the emulator. Once execution begins, the BLITZ machine will execute instructions until either an error is detected or the BLITZ machine executes a "wait" or "debug" instruction.

Here is an example:

```
> go
Beginning execution...
    < output from the BLITZ program >
Done!  The next instruction to execute will be:
000074: 01000000       wait
>
```

The "help" Command

When running the BLITZ emulator, you may type "help". Below, we show the BLITZ emulator starting up and the "help" command being executed. The "help" command may be abbreviated as "h".

```
% blitz
================================================
=====                                      =====
=====      The BLITZ Machine Emulator      =====
=====                                      =====
=====  Copyright 2001, Harry H. Porter III  =====
=====                                      =====
================================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for
info about commands.
> h
================================================================================
This program accepts commands typed into the terminal.  Each command
should be typed without any arguments; the commands will prompt for
arguments when needed.  Case is not significant.  Some abbreviations
are allowed, as shown.  Typing control-C will halt execution.

The available commands are:

  quit    - Terminate this program
  q
  help    - Produce this display
  h
  info    - Display the current state of the machine
  i
  dumpMem - Display the contents of memory
  dm
  setmem  - Used to alter memory contents
  fmem    - Display floating point values from memory
  go      - Begin or resume BLITZ instruction execution
  g
  step    - Single step; execute one machine-level instruction
  s
  t       - Single step; execute one KPL statement
  u       - Execute continuously until next call, send, or return
  stepn   - Execute N machine-level instructions
  r       - Display all the integer registers
  r1      - Change the value of register r1
   ...
  r15     - Change the value of register r15
  float   - Display all the floating-point registers
  f
  f0      - Change the value of floating-point register f0
   ...
  f15     - Change the value of floating-point register f15
  dis     - Disassemble several instructions
  d       - Disassemble several instructions from the current location
  hex     - Convert a user-entered hex number into decimal and ascii
  dec     - Convert a user-entered decimal number into hex and ascii
  ascii   - Convert a user-entered ascii char into hex and decimal
  setI    - Set the I bit in the Status Register
  setS    - Set the S bit in the Status Register
  setP    - Set the P bit in the Status Register
  setZ    - Set the Z bit in the Status Register
```

```
  setV    - Set the V bit in the Status Register
  setN    - Set the N bit in the Status Register
  clearI  - Clear the I bit in the Status Register
  clearS  - Clear the S bit in the Status Register
  clearP  - Clear the P bit in the Status Register
  clearZ  - Clear the Z bit in the Status Register
  clearV  - Clear the V bit in the Status Register
  clearN  - Clear the N bit in the Status Register
  setPC   - Set the Program Counter (PC)
  setPTBR - Set the Page Table Base Register (PTBR)
  setPTLR - Set the Page Table Length Register (PTLR)
  pt      - Display the Page Table
  trans   - Perform page table translation on a single address
  cancel  - Cancel all pending interrupts
  labels  - Display the label table
  find    - Find a label by name
  find2   - Find a label by value
  add     - Add a new label, inserting it into the indexes
  reset   - Reset the machine state and re-read the a.out file
  io      - Display the state of the I/O devices
  read    - Read a word from memory-mapped I/O region
  write   - Write a word to memory-mapped I/O region
  raw     - Switch serial input to raw mode
  cooked  - Switch serial input to cooked mode
  input   - Enter input characters for future serial I/O input
  format  - Create and format a BLITZ disk file
  sim     - Display the current simulation constants
  stack   - Display the KPL calling stack
  st
  frame   - Display the current activation frame
  fr
  up      - Move up in the activation frame stack
  down    - Move down in the activation frame stack

==========================================================================
    >
```

Abbreviated Spellings of Some Commands

Some of the commands have abbreviation, which are easier to type. Here are the abbreviations.

```
quit       q
help       h
go         g
dumpmem    dm
info       i
step       s
float      f
stack      st
frame      fr
```

The “dumpmem” Command

The “dumpmem” command (which may be abbreviated “dm”) can be used to display the contents of the BLITZ machine’s memory.

Each byte of memory is displayed in hex, with 16 bytes per line. Addresses are displayed on the left. On the right side, the same 16 bytes are displayed as ASCII, with non-printable characters displayed as periods.

Many of the emulator commands require arguments. For example, the "dumpmem" command needs a starting address and a length (in bytes). Each command should be typed on a line by itself. The emulator will then prompt for any arguments that are needed.

Here is an example:

```
> dm
Enter the starting (physical) memory address in hex: 200
Enter the number of bytes in hex (or 0 to abort): 20
00000200:  546F 0000  547F 0000  548F 0000  549F 0000    To..T...T...T...
00000210:  54AF 0000  54BF 0000  C0B0 00FF  C1B0 FF04    T...T...........
>
```

This command ignores page tables and virtual address spaces; it displays the actual (i.e., "physical") memory space.

The "setmem" Commands

The "setmem" command can be used to modify the contents of the BLITZ machine's memory.

Here is an example:

```
> setmem
Enter the (physical) memory address in hex of the word to be modified: 4c
The old value is:
0x00004C: 0xC120FF04
Enter the new value (4 bytes in hex): 123def
0x00004C: 0x00123DEF
>
```

The addresses used in this command are physical memory addresses; no page table translation is performed. (For page table translation, see the "trans" command.)

Conversion between Hex, ASCII, and Decimal

Occasionally you may need to convert between hex and decimal or see what some ASCII character code is. There are three commands that do such translations: "hex", "dec", and "ascii".

Each command asks you to enter a value. The command then prints out the value in the other two forms.

To see the ASCII code for some character, use the "ascii" command. For example:

```
> ascii
Enter a single character followed by a newline/return: h
     hex: 0x00000068     decimal: 104     ascii: "...h"
>
```

To translate a hex value into decimal, use the “hex” command.

```
> hex
Enter a value in hex: 2468abcd
     hex: 0x2468ABCD     decimal: 610839501     ascii: "$h.."
>
```

The “hex” command takes up to 4 bytes, which it will also display as four characters. You can also use it to translate a single byte.

```
> hex
Enter a value in hex: 6a
     hex: 0x0000006A     decimal: 106     ascii: "...j"
>
```

The “dec” command can be used to translate decimal numbers into hex and into ASCII characters.

```
> dec
Enter a value in decimal: 107
     hex: 0x0000006B     decimal: 107     ascii: "...k"
>
```

The “dis” Command

The emulator includes a command called “dis” which can be used to disassemble memory.

Here is a fragment of a BLITZ assembly program:

```
...
flush:
    push     r1                 ! save registers
    push     r2                 ! .
flushLoop:                      ! loop
    cleari                      !   disable interrupts
    set      outBufferCount,r1  !   r2 = outBufferCount
    load     [r1],r2            !   .
    cmp      r2,0               !   if (r2 == 0)
    be       flushLoopEx        !     break
    seti                        !   re-enable interrupts
    jmp      flushLoop          ! end
flushLoopEx:                    ! .
    seti                        ! re-enable interrupts
...
```

Assume that the BLITZ emulator is running and this program has been loaded into memory.

Below is an example of the “dis” command. In this program, it turns out the above fragment was loaded into bytes beginning at address 0x000180. Below, we ask the emulator to disassemble memory starting at that address:

```
    > dis
    Enter the beginning phyical address (in hex): 180
                        flush:
    000180: 541F0000        push    r1,[--r15]
    000184: 542F0000        push    r2,[--r15]
                        flushLoop:
    000188: 03000000        cleari
    00018C: C0100000        sethi   0x0000,r1
    000190: C1102088        setlo   0x2088,r1    ! decimal: 8328, ascii: " ."
(outBufferCount)
    000194: 6B210000        load    [r1+r0],r2
    000198: 81020000        sub     r2,0x0000,r0
    00019C: A200000C        be      0x00000C     ! targetAddr = flushLoopEx
    0001A0: 04000000        seti
    0001A4: A1FFFFE4        jmp     0xFFFFE4     ! targetAddr = flushLoop
                        flushLoopEx:
    0001A8: 04000000        seti
      ...
    >
```

The first thing to notice is that the comments from the ".s" file are lost. The second thing to notice is that the instructions are printed in greater detail than in the ".s" file.

Note that information about labels (such as "flush", "flushLoop", and "flushLoopEx") is carried in the "a.out" file. While label information is not technically part of the BLITZ program, this information is used by the emulator when disassembling, to make the result more meaningful.

Each instruction is printed both in hex and in human-readable mnemonic form. Consider the instruction:

```
    0001A4: A1FFFFE4        jmp     0xFFFFE4     ! targetAddr = flushLoop
```

This instruction is at address 0x0001a4. It is a "jump" instruction (whose opcode is hex a1) and has a relative offset of –28 (which is ffffe4 in hex). The disassembler adds the information (printed like a comment) that the value 0xffffe4 would be the instruction labeled "flushLoop".

Next, take a look at the following instruction from the ".s" file:

```
        cmp     r2,0                !   if (r2 == 0)
```

The "cmp" instruction is a "synthetic instruction". It is really assembled as a "sub" instruction which places the result in "r0". In other words, the result is discarded, although the condition codes are modified. This instruction is disassembled as:

```
    000198: 81020000        sub     r2,0x0000,r0
```

Next, look at the following instruction from the ".s" file:

```
        set     outBufferCount,r1   !   r2 = outBufferCount
```

The "set" instruction is a "synthetic instruction". It is really equivalent to 2 instructions, a "sethi" followed by a "setlo". It is disassembled as the following two lines. (The second line may wrap to a new line in this document.)

```
    00018C: C0100000       sethi   0x0000,r1
    000190: C1102088       setlo   0x2088,r1    ! decimal: 8328, ascii: " ."
(outBufferCount)
```

When the disassembler prints out values like 0x2088, it adds a comment. The comment gives the value interpreted as a decimal number, interpreted as ASCII characters, and interpreted as an address label. For this particular instruction, the label information is helpful and the decimal and ASCII information is not.

The "dis" command disassembles about 30 instructions at a time. The "d" command will disassemble another 30 instructions, beginning at whatever address the previous command ended on. Thus, by entering "d" commands repeatedly, you can disassemble a lengthy program one page at a time.

The "reset" Command

When debugging a BLITZ program, it is sometimes desirable to give up and start over. The "reset" command in the emulator will re-initialize the emulator and will re-read the "a.out" executable file. It will also reset all registers to zero, reset the state of the I/O devices, and re-read the ".blitzrc" file (if it exists) for any non-standard simulation parameters. The effect of "reset" is exactly as if the emulator had been quit and then re-started.

A typical debugging session might involve editing, compiling, and assembling a BLITZ program in one window, and running the emulator in a second window. After a bug has been found, the user would switch to the editing window and re-build the program. Then, after switching to the emulator window, the "reset" command could be used to re-read the newly built program.

Another common scenario involves trying to find a bug in a BLITZ program. Perhaps the bug has already occurred (i.e., been encountered during execution). The "reset" command could then be used to rerun the program from the beginning, in order to observe and more closely understand the bug.

The "info" Command

The command "i" (which is short for "info") can be used to dump the entire state of the CPU. Here is an example of the "i" command:

```
> i
============================
Memory size = 0x01000000     ( decimal: 16777216    )
Page size   = 0x00002000     ( decimal: 8192        )
.text Segment
    addr    = 0x00000000     ( decimal: 0           )
    size    = 0x00002000     ( decimal: 8192        )
.data Segment
    addr    = 0x00002000     ( decimal: 8192        )
    size    = 0x00002000     ( decimal: 8192        )
.bss Segment
    addr    = 0x00004000     ( decimal: 16384       )
    size    = 0x00000000     ( decimal: 0           )
=====  USER REGISTERS  =====
  r0  = 0x00000000     ( decimal: 0 )
```

```
  r1  = 0x00000000     ( decimal: 0 )
  r2  = 0x00000000     ( decimal: 0 )
  r3  = 0x00000000     ( decimal: 0 )
  r4  = 0x00000000     ( decimal: 0 )
  r5  = 0x00000000     ( decimal: 0 )
  r6  = 0x00000000     ( decimal: 0 )
  r7  = 0x00000000     ( decimal: 0 )
  r8  = 0x00000000     ( decimal: 0 )
  r9  = 0x00000000     ( decimal: 0 )
  r10 = 0x00000000     ( decimal: 0 )
  r11 = 0x00000000     ( decimal: 0 )
  r12 = 0x00000000     ( decimal: 0 )
  r13 = 0x00000000     ( decimal: 0 )
  r14 = 0x00000000     ( decimal: 0 )
  r15 = 0x00000000     ( decimal: 0 )
=====  SYSTEM REGISTERS  =====
  r0  = 0x00000000     ( decimal: 0 )
  r1  = 0x000000AA     ( decimal: 170         ascii: "...."   waitMsg )
  r2  = 0x00002088     ( decimal: 8328        ascii: ".. ."   outBufCt )
  r3  = 0x00000000     ( decimal: 0 )
  r4  = 0x00000000     ( decimal: 0 )
  r5  = 0x00000000     ( decimal: 0 )
  r6  = 0x00000000     ( decimal: 0 )
  r7  = 0x00000000     ( decimal: 0 )
  r8  = 0x00000000     ( decimal: 0 )
  r9  = 0x00000000     ( decimal: 0 )
  r10 = 0x00000000     ( decimal: 0 )
  r11 = 0x00000000     ( decimal: 0 )
  r12 = 0x00000000     ( decimal: 0 )
  r13 = 0x00000000     ( decimal: 0 )
  r14 = 0x00000000     ( decimal: 0 )
  r15 = 0x00FFFF00     ( decimal: 16776960    ascii: "...." )
=====  FLOATING-POINT REGISTERS  =====
  f0  = 0x00000000 00000000   ( value = 0 )
  f1  = 0x00000000 00000000   ( value = 0 )
  f2  = 0x00000000 00000000   ( value = 0 )
  f3  = 0x00000000 00000000   ( value = 0 )
  f4  = 0x00000000 00000000   ( value = 0 )
  f5  = 0x00000000 00000000   ( value = 0 )
  f6  = 0x00000000 00000000   ( value = 0 )
  f7  = 0x00000000 00000000   ( value = 0 )
  f8  = 0x00000000 00000000   ( value = 0 )
  f9  = 0x00000000 00000000   ( value = 0 )
  f10 = 0x00000000 00000000   ( value = 0 )
  f11 = 0x00000000 00000000   ( value = 0 )
  f12 = 0x00000000 00000000   ( value = 0 )
  f13 = 0x00000000 00000000   ( value = 0 )
  f14 = 0x00000000 00000000   ( value = 0 )
  f15 = 0x00000000 00000000   ( value = 0 )
=====================================
  PC   = 0x00000074     ( decimal: 116         ascii: "...t" )
  PTBR = 0x00000000     ( decimal: 0 )
  PTLR = 0x00000000     ( decimal: 0 )
                        ---- ----  ---- ----  ---- ----  --IS PZVN
  SR   = 0x00000034  =  0000 0000  0000 0000  0000 0000  0011 0100
           I = 1   Interrupts Enabled
           S = 1   System Mode
           P = 0   Paging Disabled
           Z = 1   Zero
           V = 0   No Overflow
           N = 0   Not Negative
==============================
```

```
  Pending Interrupts                = 0x00000008
    SERIAL_INTERRUPT
  System Trap Number                = 0x00000000
  Page Invalid Offending Address    = 0x00000000
  Page Readonly Offending Address   = 0x00000000
  Time of next timer event          = 91091
  Time of next disk event           = 2147483647
  Time of next serial in event      = 91355
  Time of next serial out event     = 2147483647
    Current Time                    = 90354
    Time of next event              = 91091
    Time Spent Sleeping             = 966
      Instructions Executed         = 89388
  Number of Disk Reads              = 0
  Number of Disk Writes             = 0
==============================
Next instruction to execute will be:
000074: 01000000       wait
>
```

Examining and Modifying Registers

The “r” command is used to display the contents of the integer registers.

```
> r
=====  SYSTEM REGISTERS  =====
  r0  = 0x00000000     ( decimal: 0 )
  r1  = 0x000000AA     ( decimal: 170         ascii: "...."   waitMsg )
  r2  = 0x00002088     ( decimal: 8328        ascii: ".. ."   outBufCt )
  r3  = 0x00000000     ( decimal: 0 )
  r4  = 0x00000000     ( decimal: 0 )
  r5  = 0x00000000     ( decimal: 0 )
  r6  = 0x00000000     ( decimal: 0 )
  r7  = 0x00000000     ( decimal: 0 )
  r8  = 0x00000000     ( decimal: 0 )
  r9  = 0x00000000     ( decimal: 0 )
  r10 = 0x00000000     ( decimal: 0 )
  r11 = 0x00000000     ( decimal: 0 )
  r12 = 0x00000000     ( decimal: 0 )
  r13 = 0x00000000     ( decimal: 0 )
  r14 = 0x00000000     ( decimal: 0 )
  r15 = 0x00FFFF00     ( decimal: 16776960    ascii: "...." )
==============================
>
```

At any instant, the BLITZ machine is either in “System Mode” or “User Mode”, as determined by the “S” bit in the status word. The BLITZ machine has two sets of registers; the “r” command will display whichever register set is in use. If the machine is in System Mode, the system registers will be displayed and if the machine is in User Mode, the user registers will be displayed.

You may also modify individual integer registers with commands such as “r3” and “r12”. For example:

```
> r1
  SYSTEM r1 = 0x000000AA     ( decimal: 170         )
Enter the new value (in hex): 123abc
  SYSTEM r1 = 0x00123ABC     ( decimal: 1194684     )
>
```

To display the contents of the floating-point registers, the "float" command (which may be abbreviated "f"), can be used:

```
> f
=====  FLOATING-POINT REGISTERS  =====
  f0  = 0x00000000 00000000   ( value = 0 )
  f1  = 0x00000000 00000000   ( value = 0 )
  f2  = 0x00000000 00000000   ( value = 0 )
  f3  = 0x00000000 00000000   ( value = 0 )
  f4  = 0x00000000 00000000   ( value = 0 )
  f5  = 0x00000000 00000000   ( value = 0 )
  f6  = 0x00000000 00000000   ( value = 0 )
  f7  = 0x00000000 00000000   ( value = 0 )
  f8  = 0x00000000 00000000   ( value = 0 )
  f9  = 0x00000000 00000000   ( value = 0 )
  f10 = 0x00000000 00000000   ( value = 0 )
  f11 = 0x00000000 00000000   ( value = 0 )
  f12 = 0x00000000 00000000   ( value = 0 )
  f13 = 0x00000000 00000000   ( value = 0 )
  f14 = 0x00000000 00000000   ( value = 0 )
  f15 = 0x00000000 00000000   ( value = 0 )
====================================
>
```

You may also modify individual floating-point registers with commands such as "f3" and "f12". For example:

```
> f5
  f5 = 0x00000000 00000000   ( value = 0 )
Enter the new value (e.g., 1.1, -123.456e-10, nan, inf, -inf): -5.7
  f5 = 0xC016CCCC CCCCCCCD   ( value = -5.7 )
>
```

The "Auto-Go" Option

Normally, the emulator begins by asking for a command. It executes the command, displays the result, and asks for the next command in a loop. The emulator can also be set to begin execution automatically. This is called the "auto-go" feature and may be specified using the command line option "-g".

```
% blitz –g
```

The "auto-go" option causes the emulator to begin executing the BLITZ program immediately. Only if errors occur, will the emulator go into command-line mode. It will display an error message and ask the user to enter a command. If the program terminates without any errors, then the emulator will also terminate.

Single-Stepping Machine Instructions

The "step" command (which may be abbreviated "s") can be used to single-step the emulator.

Entering this command will cause a single BLITZ machine instruction to be executed, and control to be returned to the emulator command interface. After the instruction is executed, the emulator will show the instruction that is due to be executed next (not the instruction that was executed) so you can stop before an instruction of interest.

In the following example, three instructions are executed.

```
> s
Done!  The next instruction to execute will be:
000078: A1FFFFE0       jmp     0xFFFFE0         ! targetAddr = busywait
> s
Done!  The next instruction to execute will be:
                   busywait:
                   loop:
000058: 6B310000       load    [r1+r0],r3
> s
Done!  The next instruction to execute will be:
00005C: 88030002       and     r3,0x0002,r0     ! decimal: 2, ascii: ".."
>
```

Executing a large number of instructions using the "step" command quickly becomes tedious. To speed up things, you can use the "stepn" command. The "stepn" instruction begins by asking you how many instructions you wish to execute. It then executes this many instructions and suspends execution.

An example of the "stepn" instruction appears next. The program being executed prints the message "Hello, world". In this example, we execute 77 instructions, which is enough to print the first part of the message. We see the characters "Hello, " displayed right after the "Beginning execution..." message.

```
> stepn
Enter the number of instructions to execute: 77
Beginning execution...
Hello, Done!  The next instruction to execute will be:
00006C: A2000010       be      0x000010         ! targetAddr = loopExit
>
```

Debugging KPL Programs

Consider this program, written in the KPL programming language. (Line numbers have been added.)

```
1    header Hello
2      uses System
3      functions
4        main ()
5    endHeader
```

```
1    code Test
2
3      function main ()
4          print ("Hello, world...\n")
5          foo ()
6        endFunction
7
8      function foo ()
9        var x: int = 1
```

```
10          bar (x + 1)
11        endFunction
12
13      function bar (a: int)
14        var y: int = a + 1
15          test (y + 1)
16        endFunction
17
18      function test (b: int)
19        var z: int = b + 1
20          print ("The value of z is ")
21          printInt (z)
22          nl ()
23          debug
24        endFunction
25
26    endCode
```

Here are the commands needed to compile, assemble, and link this program:

```
% kpl Test
% asm Test.s
% kpl System -unsafe
% asm System.s
% lddd Test.o System.o runtime.o –o Test
```

To execute the program, the emulator is invoked and the “go” command is issued:

```
% blitz Test
================================================
=====                                      =====
=====      The BLITZ Machine Emulator      =====
=====                                      =====
=====  Copyright 2001, Harry H. Porter III  =====
=====                                      =====
================================================

Enter a command at the prompt.  Type 'quit' to exit or 'help' for
info about commands.
> go
Beginning execution...
====================  KPL PROGRAM STARTING  ====================
Hello, world...
The value of z is 5

****  A 'debug' instruction was encountered  *****
Done!  The next instruction to execute will be:
0016AC: 87D00017       or      r0,0x0017,r13  ! decimal: 23, ascii: ".."
>
```

This program prints a message and then calls function “foo”. The function “foo” calls function “bar” which then calls function “test”. The function “test” prints the value of a variable and then executes the “debug” statement.

The compiler translates the KPL “debug” statement into the “debug” machine instruction. When executed, the “debug” instruction causes the emulator to immediately suspend execution, print the message

```
****  A 'debug' instruction was encountered  *****
```

and re-enter the command-line mode. The user may now inspect the state of the BLITZ machine.

The "stack" command can be used to run through the activation record stack and print information showing where, in each currently active function, execution is suspended.

```
> stack
   Function/Method        Frame Addr   Execution at...
   ===================    ==========   ====================
   test                    00FFFEA8    Test.c, line 23
   bar                     00FFFEC4    Test.c, line 15
   foo                     00FFFEE0    Test.c, line 10
   main                    00FFFEF8    Test.c, line 5
Bottom of activation frame stack!
>
```

At the top of the stack, we see that we are executing in the function "test". Furthermore, we can see the source code location of the statement being executed. The "debug" statement is on line 23 in the file named "Test.c". The function "test" was called from the function "bar" and the call occurs on line 15.

The "stack" command assumes a KPL program has been running. The command examines the contents of memory and extracts the information from the runtime stack. If memory has been corrupted, this command might print erroneous information. Nonetheless, the "stack" command is useful in debugging KPL programs.

The "frame" command can be used to see the current values of local variables. For example:

```
> frame
=====  Frame number 0 (where StackTop = 0)  =====
Function Name:    test
Filename:         Test.c
Execution now at: line 23
Frame Addr:       00FFFEA8
frameSize:        12
totalParmSize:    4
                          ==========
   sp--> -20   00FFFE94:  00000005
         -16   00FFFE98:  00000005
         -12   00FFFE9c:  000011D8
R.D.ptr:  -8   00FFFEA0:  000016C0
    r13:  -4   00FFFEA4:  0000000F
     fp:   0   00FFFEA8:  00FFFEC4
RetAddr:   4   00FFFEAc:  000015CC
                          ==========
   Args:   8   00fffeb0:  00000004

PARAMETERS AND LOCAL VARIABLES WITHIN THIS FRAME:
================================================
  b: int
           8   00FFFEB0:  00000004    value = 4
  _temp_21
         -12   00FFFE9C:  000011D8
  z: int
         -16   00FFFE98:  00000005    value = 5
================================================
>
```

In this example, the "frame" command prints information from the frame on the top of the stack, which is the frame of the currently executing function. We see the name of the function ("test") and, within it, where execution currently is (line 23 in file "Test.c"). Next, we see the exact contents of the frame in hex (between the ====== markers), as well as offsets and memory addresses.

Then we see the names of the parameters and local variables of this function, along with their types and current values. The values are given in hex. For some types of data (namely integers, doubles, Booleans, characters), the data is also printed in human-readable form. For pointers, we also see the word of data pointed to. The compiler generates temporary variables with names such as "_temp_21" and the values of these are also printed.

The stack may contain many frames. In this example, the stack contains 4 frames. When debugging some programs, we may need to look at more than just the top (currently executing) frame. To look at other frames, we use the "up" and "down" commands.

The "stack", "up", "down" commands all use a notion of the "current frame position". The "down" command moves the current position down the stack (away from the top), while the "up" command moves it back up. The "frame" command simply prints the current frame.

For example, the "down" command will take us to the frame of the function that called "test":

```
> down
=====  Frame number 1 (where StackTop = 0)  =====
Function Name:    bar
Filename:         Test.c
Execution now at: line 15
Frame Addr:       00FFFEC4
frameSize:        12
totalParmSize:    4
                          ==========
         -20   00FFFEB0:  00000004
         -16   00FFFEB4:  00000003
         -12   00FFFEB8:  00000004
R.D.ptr:  -8   00FFFEBC:  000015E0
    r13:  -4   00FFFEC0:  0000000A
     fp:   0   00FFFEC4:  00FFFEE0
RetAddr:   4   00FFFEC8:  00001514
                          ==========
   Args:   8   00FFFECC:  00000002

PARAMETERS AND LOCAL VARIABLES WITHIN THIS FRAME:
================================================
  a: int
           8   00FFFECC:  00000002    value = 2
  _temp_16
         -12   00FFFEB8:  00000004
  y: int
         -16   00FFFEB4:  00000003    value = 3
================================================
>
```

Here, we see the same information as we saw for the other frame. We see the name of the function "bar", where execution is (line 15), and the current values of the variables ("a" has value 2 and "y" has value 3).

Single-Stepping KPL Programs

Consider the "Test" program discussed above. Let's restart the program by issuing the "reset" command, which resets the CPU and reloads memory.

```
> reset
Resetting all CPU registers and re-reading file "Test"...
>
```

We can single-step the program by issuing the "t" command, which will execute a single KPL statement and re-enter the emulator's command-line mode. Here is an example showing the execution of several KPL statements:

```
> t
About to execute FUNCTION ENTRY
                                    in main (Test.c, line 3)  time = 516
> t
About to execute FUNCTION CALL (external function)
                                    in main (Test.c, line 4)  time = 523
> t
Hello, world...
About to execute FUNCTION CALL
                                    in main (Test.c, line 5)  time = 531
> t
About to execute FUNCTION ENTRY
                                    in foo (Test.c, line 8)  time = 550
> t
About to execute FUNCTION CALL
                                    in foo (Test.c, line 10)  time = 561
> t
About to execute FUNCTION ENTRY
                                    in bar (Test.c, line 13)  time = 580
> t
About to execute FUNCTION CALL
                                    in bar (Test.c, line 15)  time = 594
> t
About to execute FUNCTION ENTRY
                                    in test (Test.c, line 18)  time = 613
> t
About to execute FUNCTION CALL (external function)
                                    in test (Test.c, line 20)  time = 625
> t
The value of z is 5About to execute FUNCTION CALL
                                    in test (Test.c, line 22)  time = 642
> t
About to execute FUNCTION ENTRY
                                    in nl (System.c, line 48)  time = 655
> t
About to execute FUNCTION CALL (external function)
                                    in nl (System.c, line 49)  time = 659
> t

About to execute RETURN statement
                                    in nl (System.c, line 49)  time = 666
>
```

The "t" command executes one KPL statement and then stops. After stopping, the "t" command prints information about the next statement to be executed.

The execution of one KPL statement involves the execution of several machine instructions. Although the algorithm used by the emulator to determine exactly where the statement boundaries are is not 100% accurate, it will allow the programmer to walk through a program's execution at higher-level than machine instructions.

However, with large programs, single-stepping can get very tedious. When this happens, the programmer should consider the "u" command.

The "u" command will execute many KPL statements at once, and will stop only when a function or method is entered. The "u" command will also stop just before a return is performed.

The "u" command can be used to execute a KPL program quickly, allowing the programmer to get to the point of interest. Once there, the programmer can single-step using the "t" command, or look at the variables with the "stack" and "frame" commands.

In the above example, there were no statements other than call and return statements, so there is little reason to use the "u" command.

The next example involves the execution of a larger program, which is not shown. The "u" command is used to enter and leave different methods and functions. Once in the method called "AddToEnd", the "t" command is used to single-step execution.

```
> u
About to execute METHOD ENTRY
                  in List::IsEmpty (List.c, line 49)  time = 11087
> u
About to execute RETURN statement
                  in List::IsEmpty (List.c, line 52)  time = 11098
> u
About to execute METHOD ENTRY
                  in Thread::Yield (Kernel.c, line 290)  time = 11168
> u
About to execute FUNCTION ENTRY
                  in SetInterruptsTo (Kernel.c, line 178)  time = 11197
> u
About to execute RETURN statement
                  in SetInterruptsTo (Kernel.c, line 198)  time = 11229
> u
About to execute METHOD ENTRY
                  in List::Remove (List.c, line 33)  time = 11270
> u
About to execute RETURN statement
                  in List::Remove (List.c, line 46)  time = 11316
> u
About to execute METHOD ENTRY
                  in List::AddToEnd (List.c, line 20)  time = 11379
> t
About to execute IF statement
                  in List::AddToEnd (List.c, line 24)  time = 11381
> t
About to execute SEND
                  in List::AddToEnd (List.c, line 24)  time = 11383
> t
```

```
About to execute METHOD ENTRY
                  in List::IsEmpty (List.c, line 49)  time = 11404
> t
About to execute IF statement
                  in List::IsEmpty (List.c, line 51)  time = 11406
> t
About to execute ELSE statement
                  in List::IsEmpty (List.c, line 54)  time = 11413
> t
About to execute RETURN statement
                  in List::IsEmpty (List.c, line 54)  time = 11415
> t
About to execute THEN statement
                  in List::AddToEnd (List.c, line 25)  time = 11427
> t
About to execute ASSIGN statement
                  in List::AddToEnd (List.c, line 25)  time = 11429
> t
About to execute RETURN statement
                  in List::AddToEnd (List.c, line 24)  time = 11440
>
```

Note that the "time" displayed shows that this example spanned the execution of 353 machine instructions. The time value can be used in conjunction with the "stepn" command to quickly get to the same point again.

Next, we discuss a "trick" which allows us to effectively "back up" program execution. This can be useful when debugging. We say "effectively" back up because the CPU cannot actually be run in reverse.

Let's assume that after single-stepping the program for a while, we realize that a bug may involve something that happened a little earlier. In the above example, let's assume that we want to back up the execution and see the value of the variable "p" directly before the assignment statement is executed. In other words, we want to back-up execution to time 11429 and look at the value of "p" before the assignment statement. Unfortunately, the assignment statement has already been executed, possibly changing the value of "p."

To "back up" the CPU, we execute the "reset" command to restart the program, followed by "stepn" to get to the time of interest, followed by the "frame" command to see the variable's value.

```
> reset
Resetting all CPU registers and re-reading file "os"...
> stepn
Enter the number of instructions to execute: 11429
Beginning execution...
====================  KPL PROGRAM STARTING  ====================
Initializing Thread Scheduler...
Initializing Idle Process...
Done!  The next instruction to execute will be:
0049C4: 8B1E000C       load    [r14+0x000C],r1  ! decimal: 12
> frame
=====  Frame number 0 (where StackTop = 0)  =====
Function Name:    List::AddToEnd
Filename:         List.c
Execution now at: line 25
Frame Addr:       0002F410
```

```
    frameSize:        12
    totalParmSize:    8
                             ==========
       sp--> -20   0002F3FC:  0102D334
             -16   0002F400:  0002D334
             -12   0002F404:  00000000
    R.D.ptr:  -8   0002F408:  00004A50
        r13:  -4   0002F40C:  00000142
         fp:   0   0002F410:  0002F44C
    RetAddr:   4   0002F414:  000123D4
                             ==========
       Args:   8   0002F418:  0002D334
              12   0002F41C:  0002E404

    PARAMETERS AND LOCAL VARIABLES WITHIN THIS FRAME:
    ================================================
      self: ptr
               4   0002F414:  000123D4    --> 000123D4:  8B1EFFDC
      p: ptr
              12   0002F41C:  0002E404    --> (_P_Kernel_idleThread) 0002E404:
00011914
      _temp_19
             -12   0002F404:  00000000
      _temp_17
             -16   0002F400:  0002D334
    ================================================
```

The "fmem" Command

The "fmem" command is similar to the "dumpmem" command. Both commands display the contents of the BLITZ machine's main memory. The "dumpmem" command displays the contents in hex and in ASCII. The "fmem" command interprets the memory as holding floating-point values and print these.

Here is an example of the same block of memory. First it is displayed using the "dumpmem" command. Second it is displayed with "fmem".

```
    > dm
    Enter the starting (physical) memory address in hex: 40
    Enter the number of bytes in hex (or 0 to abort): 100
    00000040:  C010 00FF  C110 FF00  C020 00FF  C120 FF04    ........ ... ..
    00000050:  C040 0000  C140 0084  6B31 0000  8803 0002    .@...@..k1......
    00000060:  A2FF FFF8  6C54 0000  8105 0000  A200 0010    ....lT..........
    00000070:  8044 0001  6F52 0000  A1FF FFE0  0200 0000    .D..oR..........
    00000080:  A1FF FFB8  4865 6C6C  6F2C 2077  6F72 6C64    ....Hello, world
    00000090:  210A 0D00  0000 0000  0000 0000  0000 0000    !...............
    000000A0:  0000 0000  0000 0000  0000 0000  0000 0000    ................
    000000B0:  0000 0000  0000 0000  0000 0000  0000 0000    ................
    000000C0:  0000 0000  0000 0000  0000 0000  0000 0000    ................
    000000D0:  0000 0000  0000 0000  0000 0000  0000 0000    ................
    000000E0:  0000 0000  0000 0000  0000 0000  0000 0000    ................
    000000F0:  0000 0000  0000 0000  0000 0000  0000 0000    ................
    00000100:  0000 0000  0000 0000  0000 0000  0000 0000    ................
    00000110:  0000 0000  0000 0000  0000 0000  0000 0000    ................
    00000120:  0000 0000  0000 0000  0000 0000  0000 0000    ................
    00000130:  0000 0000  0000 0000  0000 0000  0000 0000    ................
    > fmem
    Enter the beginning phyical address (in hex): 40
    Dumping 256 bytes as 32 double-precision floating-points...
```

```
   000040: C01000FF C110FF00   value = -4.00097562471615
   000048: C02000FF C120FF04   value = -8.00195125129495
   000050: C0400000 C1400084   value = -32.0000230371961
busywait:
loop:
   000058: 6B310000 88030002   value = 2.18316291430363e+208
   000060: A2FFFFF8 6C540000   value = -4.19865739775962e-140
   000068: 81050000 A2000010   value = -9.56960205083827e-304
   000070: 80440001 6F520000   value = -2.22507629429519e-307
   000078: A1FFFFE0 02000000   value = -6.40656817048898e-145
   000080: A1FFFFB8 48656C6C   value = -6.40644681309642e-145
   000088: 6F2C2077 6F726C64   value = 3.3315582820848e+227
   000090: 210A0D00 00000000   value = 1.59166957876397e-149
   000098: 00000000 00000000   value = 0
   0000A0: 00000000 00000000   value = 0
   0000A8: 00000000 00000000   value = 0
   0000B0: 00000000 00000000   value = 0
   0000B8: 00000000 00000000   value = 0
   0000C0: 00000000 00000000   value = 0
   0000C8: 00000000 00000000   value = 0
   0000D0: 00000000 00000000   value = 0
   0000D8: 00000000 00000000   value = 0
   0000E0: 00000000 00000000   value = 0
   0000E8: 00000000 00000000   value = 0
   0000F0: 00000000 00000000   value = 0
   0000F8: 00000000 00000000   value = 0
   000100: 00000000 00000000   value = 0
   000108: 00000000 00000000   value = 0
   000110: 00000000 00000000   value = 0
   000118: 00000000 00000000   value = 0
   000120: 00000000 00000000   value = 0
   000128: 00000000 00000000   value = 0
   000130: 00000000 00000000   value = 0
   000138: 00000000 00000000   value = 0
>
```

Changing the Program Counter

The "setpc" command can be used to change the Program Counter (the "PC" register). The PC indicates where the next instruction will be fetched from. Changing it will, in effect, cause a branch to the new location.

```
> setpc
Please enter the new value for the program counter (PC): 40
  PC   = 0x00000040     ( decimal: 64          ascii: '@' )
Next instruction to execute will be:
000040: C01000FF       sethi   0x00FF,r1        ! 0x00FFFF00 = 16776960
>
```

Interrupt Processing

Recall that the Status Registers in the CPU contains the following bits:

```
I:   Interrupts Enabled
S:   System Mode
P:   Paging Enabled
Z:   Result is Zero
V:   Overflow Occurred
N:   Result is Negative
```

The state of these bits controls the execution behavior of the CPU; for details, consult the document titled “The BLITZ Architecture”. These bits can be changed with the following commands:

```
setI    - Set the I bit
setS    - Set the S bit
setP    - Set the P bit
setZ    - Set the Z bit
setV    - Set the V bit
setN    - Set the N bit
clearI  - Clear the I bit
clearS  - Clear the S bit
clearP  - Clear the P bit
clearZ  - Clear the Z bit
clearV  - Clear the V bit
clearN  - Clear the N bit
```

For example:

```
> sets
The S bit is now 1: System Mode.
> cleari
The I bit is now 0: Interrupts Disabled.
>
```

The contents of the Status Register is displayed as part of the “info” command. The leading 26 of the 32 bits in this registers are unused and will always be zero.

```
> info
...
                        ---- ----  ---- ----  ---- ----  --IS PZVN
  SR   = 0x0000001F  =  0000 0000  0000 0000  0000 0000  0001 1111
           I = 0   Interrupts Disabled
           S = 1   System Mode
           P = 0   Paging Disabled
           Z = 0   Not Zero
           V = 0   No Overflow
           N = 0   Not Negative
...
>
```

You may see which interrupts have been signaled with the “info” command.

```
> info
...
  Pending Interrupts                = 0x00000002
    TIMER_INTERRUPT
...
>
```

Unmaskable interrupts will be processed on the next cycle. Maskable interrupts will either be processed (if the “I” bit is set) or will remain pending until the “I” bit is changed to 1.

Pending interrupts can be cleared with the "cancel" command. This command will cancel both maskable and unmaskable interrupts.

```
> cancel
All pending interrupts have been cancelled.
>
```

When an interrupt is processed, an extra "step" cycle is required. This is illustrated in the next example. Prior to what is shown below, we assume a TIMER_INTERRUPT is pending but the "I" bit in the Status Register is "0". Thus, the interrupt is temporarily masked.

The first command is a "step", which executes the instruction directly before the "store". If we were to execute another "step" command at this point, the "store" instruction would be executed.

```
> s
Done!  The next instruction to execute will be:
000E84: 6F120000       store   r1,[r2+r0]
>
```

Instead of another "step", we issue the "seti" command, which changes the Status Register. Now the interrupt is no longer masked and interrupt processing will occur next.

```
> seti
The I bit is now 1: Interrupts Enabled.
>
```

Next, we issue a "step" command. No instruction is executed during this step cycle. Instead, the interrupt processing is initiated.

```
> s
Processing an interrupt and jumping into interrupt vector.
Done!  The next instruction to execute will be:
                  TimerInterrupt:
000004: 08000000       reti
>
```

In this program, TIMER_INTERRUPTs are dealt with by simply returning; no computation is necessary. Therefore, this program has previously loaded a return-from-interrupt instruction ("reti") into the interrupt vector in low memory, instead of a jump to the interrupt handling routine, which might be more typical of an operating system. In this program, the interrupt handler effectively consists of this single "reti" instruction.

Finally, we issue another "step" command. During this step cycle, the "reti" instruction is executed and control returns to the interrupted code.

```
> s
Done!  The next instruction to execute will be:
000E84: 6F120000       store   r1,[r2+r0]
>
```

If we were to execute another "step" command at this point, the "store" instruction would be executed.

Coping with Errors During Emulation

The BLITZ architecture describes how the BLITZ machine will respond to various instructions. Included in the BLITZ architecture is information about how various error conditions are handled. For example, an attempt to divide by zero will cause an Arithmetic Exception. Such an error will not interrupt the emulator.

In fact, all interrupts, including asynchronous hardware interrupts and synchronous exceptions during instruction execution, will be processed as specified in the BLITZ architecture. Interrupts will not stop instruction emulation.

However, there are several error conditions that the emulator will watch for. These primarily concern the I/O devices. For example, if the BLITZ program fails to fetch a character on the serial input before the next character arrives, this error will be caught by the emulator. Whenever the emulator catches an error, it will print an error message and immediately suspend instruction execution. The command loop will then be entered.

The BLITZ architecture requires word alignment on word-length data. I considered requiring double-word alignment for double-length data, just as many real machines do, but I decided not to require double alignment, since it complicates things. The presence of word alignment certainly gives the idea of alignment requirements, while having several flavors of alignment (e.g., halfword, word, double) adds little more than additional complexity.

Probabilistic and Pseudorandom Behavior

The BLITZ emulator includes the simulation of several asynchronous and probabilistic events. Most of the BLITZ architecture is deterministic, but things like interrupts will occur at random times. In addition, the emulator can simulate things like random disk read/write errors and statistical variation of timer interrupts.

In order to simulate such asynchronous or probabilistic behavior, the emulator uses a random number generator to determine when asynchronous or probabilistic events are to occur.

The random number generator supplies a sequence of pseudo-random numbers from an initial “random number seed”. The emulator uses numbers from this sequence in determining when to generate asynchronous events, etc. Since all random numbers are pseudo-random, the emulator should run exactly identically each time, as long as the initial seed is identical. Even though probabilistic behavior is being simulated, the behavior of the emulator will be fully repeatable. This is useful in debugging BLITZ programs.

To test the behavior of non-deterministic programs, you may supply a different random number seed when the emulator starts up. This will cause asynchronous events to be signaled at slightly different times. There is a default random seed which can be overridden with the “-r” command line option:

```
% blitz –r 123654
```

Each time the emulator is run with the same random seed, the results should be identical.

Unfortunately, there is a second source of non-determinism, besides the random number generator. Input for the "serial I/O" device may come from either a file or directly from the user, via the keyboard. If the input comes from a file, each run of the emulator using the same seed will be identical; there can be no variation. However, if the input comes from the keyboard, then the program may execute differently from run to run, even though the random number seed is the same. This is because the timing of the serial input interrupts will be governed by the actual arrival times of input characters from the keyboard. The BLITZ program will continue to execute (like any real CPU) waiting until the input actually arrives. The exact timing of the interrupts will be dependent on the typing of input by the user. If the BLITZ program is well-behaved, the input will be handled correctly, independently of the precise timing of keystrokes, but if the program contains race conditions, its behavior may be non-repeatable.

Emulating the BLITZ Input/Output Devices

The BLITZ computer has two I/O devices. One is a serial I/O interface and the other is a disk.

The serial I/O interface is used for communicating with a human user. The BLITZ emulator will emulate the serial I/O device by either getting input from the user (i.e., from "stdin") or by getting input from a file. If a file is used, it is specified with a command line option when the emulator is first invoked. For example:

```
% blitz –i InFileName
```

All output to the serial I/O device goes to "stdout", unless it is re-directed with the "-o" command line option:

```
% blitz –o OutFileName
```

Normally, the "-i" and "-o" options will not be used. Normally, the serial I/O will go directly to the terminal interface so the running BLITZ program will interact directly with the user.

The BLITZ disk device is emulated using a file on the host computer. All disk reads and disk writes will be simulated by getting and putting data to this file. This file is named "DISK" by default, but a different name can be given using a command line option to the emulator:

```
% blitz –d DiskFileName
```

Memory-Mapped I/O

The BLITZ architecture has no instructions specifically for I/O. Instead, the BLITZ machine communicates with various I/O devices using a technique called "memory-mapped I/O". With this approach, a region of physical memory is set aside for sending data to and receiving data from the various I/O devices of the BLITZ machine.

To send data to an external device, the CPU writes data into one of several special, predfined memory addresses. Instead of storing the bits in physical memory, the data is passed through to one of the I/O devices as described below. Likewise, to retrieve data from an external device, the CPU reads from one of several special, predefined memory locations. Instead of fetching data from main memory, the I/O

device provides data. Thus, the memory "load" and "store" commands may be used to interact with and control the external I/O devices.

Currently, there are only two devices supported by the BLITZ emulator: a serial interface and a disk drive.

Communication with the serial I/O device is through two memory addresses, called

```
SERIAL_STATUS_WORD
SERIAL_DATA_WORD
```

Communication with the disk device is through four memory addresses, called

```
DISK_STATUS_WORD
DISK_COMMAND_WORD
DISK_MEMORY_ADDRESS_REGISTER
DISK_SECTOR_NUMBER_REGISTER
```

The following constants describe where the memory-mapped region of memory is and where the various I/O addresses are located. (The values given here are the defaults. They may be changed by specifying different values in the ".blitzrc" file.)

```
MEMORY_MAPPED_AREA_LOW          0x00FFFF00
MEMORY_MAPPED_AREA_HIGH         0x00FFFFFF

SERIAL_STATUS_WORD_ADDRESS      0x00FFFF00
SERIAL_DATA_WORD_ADDRESS        0x00FFFF04

DISK_STATUS_WORD_ADDRESS        0x00FFFF08
DISK_COMMAND_WORD_ADDRESS       0x00FFFF08
DISK_MEMORY_ADDRESS_REGISTER    0x00FFFF0C
DISK_SECTOR_NUMBER_REGISTER     0x00FFFF10
DISK_SECTOR_COUNT_REGISTER      0x00FFFF14
```

Sometimes, these special, predefined memory-mapped I/O addresses are called "registers", although they are quite different from any CPU registers. Also note that the same address may be used for an input address and for an output address. (This is the case for DISK_STATUS_WORD_ADDRESS and DISK_COMMAND_WORD_ADDRESS.)

The memory-mapped region of physical memory is the range of addresses from MEMORY_MAPPED_AREA_LOW to MEMORY_MAPPED_AREA_HIGH, inclusive. Normally it is 256 bytes, as shown above.

All addresses in the memory-mapped region besides those mentioned above for the serial I/O device and the disk device are unassigned and should not be used. Any attempt to load or store from those addresses will be caught by the emulator. An error message will be printed and instruction emulation will be suspended.

Note that these constants are shared by both the emulator and the program being emulated. A change to one of these values would require a change to both the BLITZ emulator as well as the BLITZ program itself.

The Serial I/O Device

The BLITZ computer includes a serial I/O device, which allows BLITZ programs to communicate with the outside world via a two-way, asynchronous stream of bytes. The serial device is also referred to as the "terminal" device.

The serial I/O interface is intended to be a simplified model of a typical interface to a standard UART serial interface chip, which in turn interfaces to something like an RS-232 terminal or modem port.

The serial I/O interface might be connected to either a display terminal (such as an old-fashioned teletype terminal, which transmits and receives ASCII characters, one-by-one), or to a modem, or to an RS-232 type serial interface. With the BLITZ emulator, the serial I/O interface is connected to either the terminal you are using (e.g., Unix "stdin" and "stdout") or to a file (using the –i and –o options on the emulator command line). By using stdin and stdout, you can communicate with a running BLITZ program simply by typing on the terminal.

The serial I/O interface is asynchronous and two-way, which means that bytes may be transmitted in either direction simultaneously, with no timing connection between the input and output flows.

The communication is through two memory-mapped registers, called

```
SERIAL_STATUS_WORD
SERIAL_DATA_WORD
```

At any moment, the serial I/O device is either busy sending a character or not, and it is busy receiving a character or not. To determine the status of the device, a BLITZ program may read from the SERIAL_STATUS_WORD location in memory. The word retrieved will have this format:

```
  byte 1      byte 2      byte 3      byte 4
==== ====   ==== ====   ==== ====   ==== ====
0000 0000   0000 0000   0000 0000   0000 00RA

R = OutputReady bit (1=ready, 0=not ready)
A = CharacterAvailable bit (1=available, 0=not available)
```

When the device is ready and capable of sending a new character to the terminal, the OutputReady bit will be 1. To start the transmission of a character to the terminal, the BLITZ program should write a word to the SERIAL_DATA_WORD. The least significant byte of this word should contain a byte of data, which will normally be an ASCII character. (The remaining bytes in word are ignored.) The OutputReady bit will change to a zero, and the character will be transmitted. The transmission is not instantaneous, but is in fact a rather slow process, so the OutputReady bit will stay zero for some time. Later, after the transmission is completed, the device will become ready to receive another character for transmission, and the OutputReady bit will once again change to 1.

From time-to-time keys may be pressed on the keyboard (or bytes will be received on the serial I/O interface). Each time a key is pressed, the CharacterAvailable bit will change to 1. The BLITZ program may then get the character by reading from the SERIAL_DATA_WORD. Whenever the BLITZ program reads the SERIAL_DATA_WORD, the CharacterAvailable bit will change to 0. It is the BLITZ program's responsibility to retrieve the characters from the SERIAL_DATA_WORD in a timely way; if the data is not retrieved, it will be lost when the next character comes in from the keyboard. (It is

not an error to re-read from the SERIAL_DATA_WORD, before CharacterAvailable becomes true again for the next character.)

Every time a transmission is completed and the OutputReady bit is changed to 1, a SerialInterrupt will occur. Also, every time a character reception is completed and the CharacterAvailable bit is changed to 1, a SerialInterrupt will occur. (However, when these bits are changed to zero, there will be no interrupt.) The BLITZ program may read from the SERIAL_STATUS_WORD as often as desired to check the state of the bits. The program should never write to the SERIAL_STATUS_WORD.

The emulator checks for several errors that may occur regarding the proper operation of the serial I/O device. If the BLITZ program writes to the SERIAL_STATUS_WORD, an error message will be displayed and instruction execution will be suspended immediately. If the BLITZ program attempts to send a character to the terminal (by writing to the SERIAL_DATA_WORD) before the terminal is ready to display the next character (i.e., while OutputReady is false), an error message will be displayed and instruction execution will be suspended immediately. If a character is input (i.e., a key is pressed), before the previous input character was retrieved (i.e., before the BLITZ program has read from the SERIAL_DATA_WORD), then an error message will be displayed and instruction execution will be suspended immediately.

Echoing and Buffering of Raw and Cooked Serial Input

With an operating system such as Unix, some rather complex processing is done on character input and output to a terminal. For example, whenever the user hits a key on the keyboard, the character is normally echoed by Unix and then simply added to a buffer area. The characters in the buffer are accumulated as they are typed, but are not given to the user-level program until the user hits the ENTER key. Then, the entire line of characters is given to the user-level program all at once.

Unix also handles several control characters specially. For example, when the user hits the backspace key, Unix will send characters to the terminal to back up the cursor, display a blank to over-write the previous character, and finally reposition the cursor.

When the user hits the ENTER key, many terminals will transmit the ASCII “CR” character, not the ASCII “NL” character to the computer. Recall that the CR and NL characters are different.

	ASCII name	C notation	Hex value	Decimal value
“newline”	NL (or LF)	‘\n’	0A	10
“return”	CR	‘\r’	0D	13

Unix generally echoes the CR character with two characters: the CR followed by the NL. Then, an NL character is added to the buffer, instead of the CR character which was actually received.

With Unix, all of this processing is completely configurable, making it possible to use many different types of terminal, each with slightly different behaviors, while not requiring any change to user-level programs. For example, all programs expect lines to end with NL, even though some terminals may send different characters when the ENTER key is pressed.

When the BLITZ emulator uses “stdin” and “stdout” for the serial I/O device, the emulator may run in either of two modes: “raw” or “cooked”. The default is “cooked” mode, but you may switch the emulator between modes with the “raw” and “cooked” commands.

```
> raw
Future terminal input will be "raw".
> cooked
Future terminal input will be "cooked".
>
```

You may also use the command line option “-raw” to put the emulator in “raw” mode. This is particularly useful when running with the “auto-go” command line option.

Raw mode is intended to allow the emulator to function more exactly like a real computer. All buffering, echoing, and special processing of certain control characters is left to the BLITZ program. The Unix I/O processing is turned off and the emulator simply passes the keystrokes through to the BLITZ serial I/O device. The BLITZ program must perform all buffering, echoing, and special processing required. If the BLITZ program fails to echo characters, it may appear that your computer is dead, since it does not seem to respond to keystrokes. Also, the BLITZ program must deal with any differences in different types of terminals. The terminal you are using may supply a “CR”, an “LF”, or some other character when ENTER is pressed; the BLITZ program must be able to handle each. The bottom line is that this puts a lot of extra work on the BLITZ program.

In “cooked” mode, the emulator runs like most normal Unix user-level programs, making use of all the special terminal configuration software included in Unix. This allows you to use whichever terminal you use normally, without having to put terminal-specific code into your BLITZ program. Whenever you type input to be supplied to the BLITZ serial I/O interface, it is buffered, echoed, and processed by Unix first. For example, you may correct typing errors with the backspace key and the BLITZ program will see only the corrected data. In cooked mode, you must type a full line, followed by ENTER before the BLITZ program will see any characters at all. The BLITZ program will always sees a single NL character at the end of every line of input.

To accurately model a real OS, your BLITZ program should echo all characters received on the serial input. When you are using cooked input, this has the effect of causing a double echoing of input: First the Unix terminal drivers will echo the characters as you type them. Then, at the end of the line when you type ENTER, all characters will be supplied one at a time to the serial I/O device, and the BLITZ program will then (presumably) echo each character. Thus, the line just entered will be redisplayed a second time, if the BLITZ program is echoing its input properly.

When debugging programs that process serial input using interrupts, it is useful to be able to control exactly which characters are read from the serial device. However, it is difficult to type input to the BLITZ computer while also entering commands to the emulator. To alleviate this problem, you may use the “input” command, which allows you to type ahead several characters, which will be supplied to the serial interface when it is ready.

```
> input
The following characters will be supplied as input to the BLITZ serial
input before querying the terminal:
  ""
You may add to this type-ahead buffer now.
The terminal is now in "cooked" mode.
Enter characters followed by control-D...
abc
def
^D
The following characters will be supplied as input to the BLITZ serial
input before querying the terminal:
  "abc\ndef\n"
>
```

To determine the status of the serial I/O device, the “io” command may be used. This command displays information about the serial device, the disk device, and some information about the status of the CPU.

```
> io
==========  Serial I/O  ==========
  Output Status:       Ready
  Input Status:        Character Not Available
  Current Input Char:  '\0'    (already fetched by CPU)
    The following characters are currently in the type-ahead buffer:
      "abc\ndef\n"
  Input coming from: stdin
    Input Mode: Cooked
==========  Disk I/O  ==========
...
>
```

The “wait” Instruction

The BLITZ architecture includes an instruction called “wait”. This instruction will suspend further instruction execution and the CPU will go into a low-power wait/sleep state. The only thing that will cause instruction execution to resume is an interrupt. If no interrupts occur the CPU will remain forever dormant.

How does the emulator handle the “wait” instruction? When does the emulator suspend emulation and return to the command interface?

The emulator handles the “wait” instruction as follows: If there is disk activity that is not yet complete or serial output activity that is not yet complete, then the emulator will continue until the activity is completed. Otherwise, the emulator may suspend emulation, depending on the status of the serial input.

If serial input is coming from “stdin” and interrupts are enabled, then the emulator will wait for user input and then will continue execution by signaling an interrupt. If serial input is coming from “stdin” but interrupts are disabled, then the emulation will halt. If input is coming from a file and we have reached the end of file, then emulation will halt. Otherwise, if there is more left in the input file, emulation will continue.

Note that the emulator will ignore timer interrupts in determining whether to halt emulation. There will always be another timer interrupt, so timer interrupts would keep emulation going forever and emulation

would never halt if the emulator paid attention to timer events. In other words, if execution is suspended on a "wait" instruction and the only thing that could cause an interrupt is a timer event, then emulation will be suspended.

(Note that this may cause difficulties in certain kinds of programs. Imagine a program that simply counts timer interrupts in order to wait a certain amount of time, and then prints a message after (say) 10 interrupts. The program begins by initializing a counter then performs a "wait". This program will not be emulated correctly, since the emulator will suspend emulation after the "wait" is encountered.)

The Disk Device

A BLITZ computer includes a disk drive, and the BLITZ emulator simulates a virtual disk drive.

Real disk drives store bytes in sectors. For example a sector might have 8K bytes. The sector is the minimum unit of data transfer. The main operations are "read a sector" and "write a sector". Sectors are arranged in tracks. Each track traces out a concentric circle on a rotating magnetic platter.

Generally a disk has several platters rotating together on one axis. Consequently, tracks are arranged in cylinders. For example, a disk with 5 platters and a read/write head on each side of each platter would have 10 tracks per cylinder, All 10 of the read/write heads are attached to a single, comb-like arm, so all 10 heads move together. To read any sector within a single cylinder, no arm movement is necessary if the arm is already positioned on the correct cylinder.

To read or write a sector, the read-write heads must be moved to the correct cylinder. This is called the "seek time". The seek involves physically moving the arm, the time of which is proportional to the length moved. After the movement, a "settle" time occurs, during which the vibrations in the arm created by the movement die out. In addition, the disk platter is constantly rotating, so an additional delay involves waiting until the desired sector comes under one of the read/write heads. This is called the "rotational delay". Finally, the data is transferred at a constant rate determined by how fast the disk is spinning. This is called the "transfer time" or "transfer rate" and is measured in bytes per second.

Often disks are described using "transfer rate" and "average access time". The "access time" is the sum of the seek time, the settle time, and the rotational delay, and will vary from operation to operation depending on where the disk heads are before the operation.

In the virtual disk provided by the BLITZ emulator, things are simplified. First, there is only one platter and only one side is used; in other words, there is just one track per cylinder. Consequently, we view the disk as an integral number of tracks. Each track contains a number of sectors, numbered from zero up to some maximum sector number, given by SECTORS_PER_TRACK.

The size of each sector is identical to the page size in the machine, which is 8K bytes (i.e., 8192 bytes).

The actual disk data is kept in a separate file on the host system. Normally, this file is named "DISK" and is opened when the BLITZ emulator begins. The size of the virtual disk will be an integral number of tracks. The actual number of tracks will be determined based on the size of the DISK file, when the emulator starts up. A different filename (other than "DISK") may be specified with the "-d filename" command line option to the BLITZ emulator.

The DISK file can be created using an emulator command called “format”. The format command will ask for the desired number of tracks. It will then create and initialize the file. This command can also be used to change the size of the file.

The format of data stored in the DISK file should not be of concern to the BLITZ programmer, but it consists of a 4 byte “magic number” at the beginning of the file, followed by N sectors of data bytes. Thus, it has the following format:

```
Size  Description
====  ============================================
  4   Magic number 0x53504B64 (ASCII code for "BLZd")
8192  Sector 0
8192  Sector 1
8192  Sector 2
   ...
8192  Sector K-1
```

We can summarize the virtual disk as follows:

```
Filename:                    "DISK"
Number of Tracks per Disk:    <variable>
Number of Sectors per Track: 16
Number of Bytes per Sector:  8K (8192 bytes)
```

We can measure the size of the disk in tracks:

```
NUMBER_OF_TRACKS
```

or in sectors:

```
NUMBER_OF_SECTORS = NUMBER_OF_TRACKS * SECTORS_PER_TRACK
```

or in bytes:

```
NUMBER_OF_BYTES = NUMBER_OF_SECTORS * BYTES_PER_SECTOR
```

For example, a typical DISK file might have the following size:

```
NUMBER_OF_TRACKS = 100
NUMBER_OF_SECTORS = 1,600
NUMBER_OF_BYTES = 13,107,200
```

The sectors on the disk are number from 0 up to the maximum:

```
0, 1, 2, ... , NUMBER_OF_SECTORS-1
```

The basic operations that the BLITZ programmer can do are:

```
Read K Sectors into Memory
Write K Sectors from Memory
```

The disk is controlled by reading and writing the following memory-mapped I/O registers. Each is a 32-bit word in the memory-mapped I/O region of physical memory.

```
DISK_STATUS_WORD
DISK_COMMAND_WORD
DISK_MEMORY_ADDRESS_REGISTER
DISK_SECTOR_NUMBER_REGISTER
DISK_SECTOR_COUNT_REGISTER
```

The disk is either busy reading or writing, or is free and available. The DISK_STATUS_WORD may be read at any time. The following values indicate the status of the disk and the result of the last disk operation.

```
DiskBusy                        0x00000000
OperationCompletedOK            0x00000001
OperationCompletedWithError1    0x00000002
OperationCompletedWithError2    0x00000003
OperationCompletedWithError3    0x00000004
OperationCompletedWithError4    0x00000005
OperationCompletedWithError5    0x00000006
```

The following commands may be written to the DISK_COMMAND_REGISTER:

```
DiskReadCommand    0x00000001
DiskWriteCommand   0x00000002
```

The “read” operation will transfer 1 or more sectors from the disk into memory. To perform a read operation, the BLITZ program must take the following steps.

First, the DISK_SECTOR_NUMBER_REGISTER, DISK_SECTOR_COUNT_REGISTER and the DISK_MEMORY_ADDRESS_REGISTER must be loaded (in any order). Then the program must move the “DiskReadCommand” to the DISK_COMMAND_WORD.

The number of sectors to be transferred must be placed in the DISK_SECTOR_COUNT_REGISTER. This should be between 1 and NUMBER_OF_SECTORS. The sector from the disk should then be loaded into the DISK_SECTOR_NUMBER_REGISTER. This number must be between 0 and NUMBER_OF_SECTORS-1. It is an error to attempt to read or write beyond the end of the disk. The DISK_MEMORY_ADDRESS should be loaded with the physical memory address into which to place the data. It is an error to attempt transfer data to/from any address which is not in physical memory. In addition, you may not transfer data to/from any address in the memory-mapped I/O area.

After the command has been issued, the disk will become busy for some time. When the operation is finished, the status will change to either “Operation Completed OK” or “Operation Completed with Error”.

The “write” operation is very similar to the “read” operation: First, the following registers should be loaded:

```
DISK_MEMORY_ADDRESS_REGISTER
DISK_SECTOR_NUMBER_REGISTER
DISK_SECTOR_COUNT_REGISTER
```

in any order. Then the “DiskWriteCommand” command should be written to the DISK_COMMAND_WORD. The disk will be busy while the data is moved from the memory to the disk. Then the disk status will change to “Operation Completed OK” or “Operation Completed with Error”.

After issuing a DiskReadCommand or a DiskWriteCommand, the disk will become busy for a while. When the operation completes (either normally or with an error), a DiskInterrupt will occur and the disk status will change to OperationCompletedOK or OperationCompletedWithError.

The following are considered errors and will result in the DISK_STATUS_WORD being "OperationCompletedWithError". The status will remain unchanged until the next "read" or "write" operation is initiated (i.e., until the DISK_COMMAND_WORD is written to).

(Error 1) The MEMORY_ADDRESS_REGISTER is not aligned on a memory page boundary. The last 13 bits of the REGISTER should always be zeros. Also, the SECTOR_COUNT_REGISTER is not positive.

(Error 2) The MEMORY_ADDRESS_REGISTER and length specification include memory addresses that are not legal physical addresses or that are in the memory-mapped I/O area.

(Error 3) The DISK_SECTOR_NUMBER_REGISTER and length specification include a sector that is not between 0 and NUMBER_OF_SECTORS-1.

(Error 4) The DISK had some sort of a I/O error. This could be a "soft error", in which case the operation will succeed if re-tried, or it could be a "hard error", in which case the operation will never succeed if re-tried. It is assumed that the disk controller itself has re-tried the operation a few times. Therefore, any errors reported here can be assumed to be "hard" errors. It is better for the BLITZ program to print a message for the user and give up when I/O errors occur. Perhaps the user can correct the error (e.g., by reconnecting a disk cable) and re-try the operation later.

In a real disk, each sector is written with a header and trailer, and error-checking codes are computed and written on the disk, in addition to the actual data bytes. In a most disk drives, headers and trailers are handled entirely by the disk controller, not by the CPU. The BLITZ system emulates such an approach, assuming headers and trailers are handled by the device controller. Consequently the details of the header, trailer, and error-checking codes are not specified. The BLITZ program only needs to concern itself with the actual bytes of data, not with headers, trailers, or error-checking codes.

Several things can go wrong during a disk operation and there can be "soft" or "hard" errors during and disk operation. For example, a disk read operation may have a failure in the error-checking process; this would be a soft error, since it will generally disappear upon a re-try. Or the disk may be disconnected or the power may be turned off, which would be a hard error.

(Error 5) Invalid Command Word. The program has attempted to store something besides DiskReadCommand or DiskWriteCommand into the command register.

In addition to checking for the errors listed above, the BLITZ emulator also performs additional error checking on the use of the disk device by the BLITZ program. The emulator checks to make sure that the DISK_SECTOR_NUMBER_REGISTER, DISK_MEMORY_ADDRESS_REGISTER, and DISK_SECTOR_COUNT_REGISTER are each loaded exactly once before each read or write operation. It also checks to make sure that bytes in the memory buffer being transferred to or from disk are not accessed by the CPU while the disk operation is still in progress. If any errors like this are detected, a message is displayed and BLITZ instruction execution is immediately suspended.

When a DISK file is created or enlarged by the "format" command, the data in the file must be initialized. The "format" command will initialize all new sectors with ASCII character data giving the sector number. The data will consist of a pattern of repeating characters. For example, assume that a DISK file with just 1 track (16 sectors) is created. Here is how the file would be initialized:

```
00000000:  5350 4B64  3C2D 2D2D  4245 4749  4E4E 494E    BLZd<---BEGINNIN
00000010:  4720 4F46  2053 4543  544F 522D  2D2D 2D2D    G OF SECTOR-----
00000020:  2D2D 2D2D  2D2D 2D2D  2D2D 6469  736B 2073    ----------disk s
00000030:  6563 746F  7220 3030  3030 3030  2064 6973    ector 000000 dis
00000040:  6B20 7365  6374 6F72  2030 3030  3030 3020    k sector 000000
00000050:  6469 736B  2073 6563  746F 7220  3030 3030    disk sector 0000
00000060:  3030 2064  6973 6B20  7365 6374  6F72 2030    00 disk sector 0
00000070:  3030 3030  3020 6469  736B 2073  6563 746F    00000 disk secto
00000080:  7220 3030  3030 3030  2064 6973  6B20 7365    r 000000 disk se
00000090:  6374 6F72  2030 3030  3030 3020  6469 736B    ctor 000000 disk
000000A0:  2073 6563  746F 7220  3030 3030  3030 2064     sector 000000 d
000000B0:  6973 6B20  7365 6374  6F72 2030  3030 3030    isk sector 00000
000000C0:  3020 6469  736B 2073  6563 746F  7220 3030    0 disk sector 00
000000D0:  3030 3030  2064 6973  6B20 7365  6374 6F72    0000 disk sector
000000E0:  2030 3030  3030 3020  6469 736B  2073 6563     000000 disk sec
000000F0:  746F 7220  3030 3030  3030 2064  6973 6B20    tor 000000 disk
00000100:  7365 6374  6F72 2030  3030 3030  3020 6469    sector 000000 di
00000110:  736B 2073  6563 746F  7220 3030  3030 3030    sk sector 000000
00000120:  2064 6973  6B20 7365  6374 6F72  2030 3030     disk sector 000
  ...
00001FA0:  3020 6469  736B 2073  6563 746F  7220 3030    0 disk sector 00
00001FB0:  3030 3030  2064 6973  6B20 7365  6374 6F72    0000 disk sector
00001FC0:  2030 3030  3030 3020  6469 736B  2073 6563     000000 disk sec
00001FD0:  746F 7220  3030 3030  3030 2D2D  2D2D 2D2D    tor 000000------
00001FE0:  2D2D 2D2D  2D2D 2D2D  2D2D 2D2D  2D2D 2D2D    ----------------
00001FF0:  2D2D 2D45  4E44 204F  4620 5345  4354 4F52    ---END OF SECTOR
00002000:  2D2D 2D3E  3C2D 2D2D  4245 4749  4E4E 494E    --->&lt;---BEGINNIN
00002010:  4720 4F46  2053 4543  544F 522D  2D2D 2D2D    G OF SECTOR-----
00002020:  2D2D 2D2D  2D2D 2D2D  2D2D 6469  736B 2073    ----------disk s
00002030:  6563 746F  7220 3030  3030 3031  2064 6973    ector 000001 dis
00002040:  6B20 7365  6374 6F72  2030 3030  3030 3120    k sector 000001
00002050:  6469 736B  2073 6563  746F 7220  3030 3030    disk sector 0000
00002060:  3031 2064  6973 6B20  7365 6374  6F72 2030    01 disk sector 0
  ...
0001FF10:  6563 746F  7220 3030  3030 3135  2064 6973    ector 000015 dis
0001FF20:  6B20 7365  6374 6F72  2030 3030  3031 3520    k sector 000015
0001FF30:  6469 736B  2073 6563  746F 7220  3030 3030    disk sector 0000
0001FF40:  3135 2064  6973 6B20  7365 6374  6F72 2030    15 disk sector 0
0001FF50:  3030 3031  3520 6469  736B 2073  6563 746F    00015 disk secto
0001FF60:  7220 3030  3030 3135  2064 6973  6B20 7365    r 000015 disk se
0001FF70:  6374 6F72  2030 3030  3031 3520  6469 736B    ctor 000015 disk
0001FF80:  2073 6563  746F 7220  3030 3030  3135 2064     sector 000015 d
0001FF90:  6973 6B20  7365 6374  6F72 2030  3030 3031    isk sector 00001
0001FFA0:  3520 6469  736B 2073  6563 746F  7220 3030    5 disk sector 00
0001FFB0:  3030 3135  2064 6973  6B20 7365  6374 6F72    0015 disk sector
0001FFC0:  2030 3030  3031 3520  6469 736B  2073 6563     000015 disk sec
0001FFD0:  746F 7220  3030 3030  3135 2D2D  2D2D 2D2D    tor 000015------
0001FFE0:  2D2D 2D2D  2D2D 2D2D  2D2D 2D2D  2D2D 2D2D    ----------------
0001FFF0:  2D2D 2D45  4E44 204F  4620 5345  4354 4F52    ---END OF SECTOR
00020000:  2D2D 2D3E                                    --->
```

The "io" Command

The current status of the serial I/O device and the disk device can be seen with the "io" command. For example:

```
> io
==========  Serial I/O  ==========
  Output Status:       Ready
  Input Status:        Character Not Available
  Current Input Char:  '\0'    (already fetched by CPU)
    The following characters are currently in the type-ahead buffer:
      ""
  Input coming from: stdin
    Input Mode: Cooked
==========  Disk I/O  ==========
  The file used for the disk: "DISK"
    DISK File is currently opened.
  Disk size:
    Total Tracks = 3
    Total Sectors = 48
    Sectors per track = 16
  Current Status:
    Positioned at Sector = 0
    Current Disk Status  = OPERATION_COMPLETED_OK
    Future Disk Status   = OPERATION_COMPLETED_OK
  Area of memory being read from / written to:
    diskBufferLow  = 0x00000000
    diskBufferHigh = 0x00000000
  Memory-Mapped Register Contents:
    DISK_MEMORY_ADDRESS_REGISTER = 0x00000000
    DISK_SECTOR_NUMBER_REGISTER  = 0x00000000
    DISK_SECTOR_COUNT_REGISTER   = 0x00000000
  Number of Disk Reads  = 0
  Number of Disk Writes = 0
==============================
  CPU status:
    Interrupts:   Disabled
    Mode:         System
    Pending Interrupts:
      TIMER_INTERRUPT
  Time of next timer event........ 1001
  Time of next disk event......... 2147483647
  Time of next serial in event.... 0
  Time of next serial out event... 2147483647
    Current Time.................. 0
    Time of next event............ 0
==============================
>
```

The I/O devices are activated by reading and writing words in the "memory-mapped" region of physical memory.

You can do this from the command line in the emulator using the "read" and "write" commands. For example, you can retrieve the status of the serial I/O device by reading from the SERIAL_STATUS_WORD (at address 0x00ffff00) as follows:

```
> read
This command can be used to read a word of memory that is in the
    memory-mapped I/O region, retrieving I/O device status or data.
Enter the (physical) memory address in hex of the word to be
read from: ffff00
Reading word...   address = 0xFFFF00    value = 0x00000002
>
```

The value is "0x00000002", which indicates that the output is ready to receive a character but that no character is available on the input.

You can write a value to a memory-mapped word using the "write" command. If we write to the SERIAL_DATA_WORD (i.e., address 0x00ffff04), it will cause a character to be written. In the following example, we write a word containing the ASCII code for the letter "a", which is 0x61. Notice that there is an "a" printed out immediately.

```
> write
This command can be used to write to a word of memory that is in the
  memory-mapped I/O region, sending data or commands to the I/O device.
Enter the (physical) memory address in hex of the word to be
written to: ffff04
Enter the new value (4 bytes in hex): 00000061
aWriting word...   address = 0xFFFF04    value = 0x00000061
>
```

The ".blitzrc" File: Changing Emulation Parameters

There are a number of simulation parameters that may be changed. The parameters that can be changed are listed below, along with their default values.

```
KEYBOARD_WAIT_TIME                30000
KEYBOARD_WAIT_TIME_VARIATION        100
TERM_OUT_DELAY                      100
TERM_OUT_DELAY_VARIATION             10
TIME_SLICE                         5000 (0=no timer interrutps)
TIME_SLICE_VARIATION                 30
DISK_SEEK_TIME                    10000
DISK_SETTLE_TIME                   1000
DISK_ROTATIONAL_DELAY               100
DISK_ACCESS_VARIATION                10
DISK_READ_ERROR_PROBABILITY         500 (0=never,1=always,n="about 1/n")
DISK_WRITE_ERROR_PROBABILITY        500 (0=never,1=always,n="about 1/n")
INIT_RANDOM_SEED             1829742401
MEMORY_SIZE                  0x01000000 (decimal: 16777216)
MEMORY_MAPPED_AREA_LOW       0x00FFFF00
MEMORY_MAPPED_AREA_HIGH      0x00FFFFFF
SERIAL_STATUS_WORD_ADDRESS   0x00FFFF00
SERIAL_DATA_WORD_ADDRESS     0x00FFFF04
DISK_STATUS_WORD_ADDRESS     0x00FFFF08
DISK_COMMAND_WORD_ADDRESS    0x00FFFF08
DISK_MEMORY_ADDRESS_REGISTER 0x00FFFF0C
DISK_SECTOR_NUMBER_REGISTER  0x00FFFF10
DISK_SECTOR_COUNT_REGISTER   0x00FFFF14
```

When the emulator starts up, it looks to see if a file named ".blitzrc" exists. If it does not, then the default values are used. If ".blitzrc" is found, then it will contain values which are read in and used

instead of the defaults. (The emulator will also re-read the ".blitzrc" file when the "reset" command is issued.)

The emulator command "sim" can be used to view the current settings, and produces a display like the list above. The "sim" command may also be used to create a new ".blitzrc" file; this is convenient since the user can then edit the file to modify one or two of the values as necessary.

The "sim" command will ask if you wish to create a new ".blitzrc" file. If you say "yes", then it will write out a new file using the current values. Here is what the "sim" command looks like:

```
> sim
====================  Simulation Constants  ========================
  KEYBOARD_WAIT_TIME                  30000
  KEYBOARD_WAIT_TIME_VARIATION          100
    ...
  DISK_SECTOR_NUMBER_REGISTER   0x00FFFF10
  DISK_SECTOR_COUNT_REGISTER    0x00FFFF14
======================================================================

The simulation constants will be read in from the file ".blitzrc"
if it exists when the emulator starts up.  If the file does not
exist at startup, defaults will be used.  You may edit the ".blitzrc"
file to change the values and then restart the emulator.

Would you like me to write these values out to the file ".blitzrc" now?
```

If the user answers "yes", then a file will be created. The file will look like this:

```
! BLITZ Simulation Constants
!
! This file is read by the BLITZ emulator when it starts up and after a
! "reset" command.  This file is used to initialize various values that
! will be used by the emulator.
!
! This file was produced by the emulator (with the "sim" command). It may
! be edited to change any or all values.
!
! Each line has variable name followed by an integer value.  A value may
! be specified in either decimal (e.g., "1234") or hex (e.g.,
! "0x1234abcd").  Values may be left out if desired, in which case a
! default will be used.  In the case of the random seed, any value
! specified here will override a value given with a command line
! option (-r).
!
!
KEYBOARD_WAIT_TIME                  30000
KEYBOARD_WAIT_TIME_VARIATION          100
  ...
DISK_SECTOR_NUMBER_REGISTER   0x00FFFF10
DISK_SECTOR_COUNT_REGISTER    0x00FFFF14
```

The Page Table Commands

At any time, these two BLITZ registers describe a page table:

```
PTBR: Page Table Base Register
PTLR: Page Table Length Register
```

The “pt” command will print out the current page table. It will use the current values of these registers to find the bytes in memory and will then interpret them as a page table.

Here is an example of the “pt” command.

```
> pt
  Page Table:
    base   (PTBR) = 0x00001000
    length (PTLR) = 0x0000000C
    This table describes a logical address space with 0x00006000
                                (decimal 24576) bytes (i.e., 3 pages)

    addr      entry        Logical  Physical  Undefined Bits  Dirty
  ========   ========     ========  ========  ==============  =====
  00001000:  00344003     00000000  00344000       000           0
  00001004:  00346007     00002000  00346000       000           0
  00001008:  0036200F     00004000  00362000       000           1

                                          Referenced  Writeable  Valid
                                          ==========  =========  =====
                                              0           1        1
                                              1           1        1
                                              1           1        1
>
```

The “pt” command prints out a single line for each page table entry. In this example, the table has 3 entries. (In this document, the display will not fit on one line and it is has been reformatted to fit the available space.)

The format and operation of the page table entries is described in the BLITZ architecture document.

For each page table entry, the “pt” command displays the actual 4 byte entry under the heading “entry”. On the same line, the command also displays the interpretation of these bits under the headings “Logical”, “Physical”, “Undefined Bits”, “Dirty”, “Referenced”, “Writeable”, and “Valid”.

Occasionally it is useful to see how some particular logical (or “virtual”) address will be interpreted using the current page table. The “trans” command can be used for this.

The “trans” command will ask for a logical address. It will then consult the page table to determine which physical address will be accessed. When the page table is used during program execution, it may be updated. The “trans” command will ask whether or not you wish to update the page table. Exceptions may also occur, and the “trans” command will ask whether you wish to cause a excecption.

Here is an example of the “trans” command. First, we use the “setp” command to turn on page table translation.

```
> setp
The P bit is now 1: Paging Enabled.
Next instruction to execute will be:
344000: 00000000       nop
```

Next, we issue the "trans" command. We supply a virtual address of 0x000468. We are interested in reading this word, without update, so we answer "yes" to the question about "read-only". Such an operation would set the "Referenced" bit, but would not set the "Dirty" bit. However, we do not want to actually modify the page table, so we answer "no" to the last question.

```
> trans
Please enter a logical address: 468
Will this be a read-only operation (y/n)? y
After figuring out the affect of this memory access, do you want me to
update the page table and signal exceptions, if any, as if this
operation were performed (y/n)? n
Calling:
    translate (logicalAddr=0x00000468, reading=TRUE,
                          wantPrinting=TRUE, doUpdates=FALSE)
*****  PAGE TRANSLATION BEGINNING  *****
   Logical address             = 0x00000468
     Page Number               = 0x00000000
     Offset into page          = 0x00000468
   Status[P] = 1, Paging is turned on
   Page Table:
            base               = 0x00001000
            length             = 0x0000000C
            addr of last entry = 0x00001008
   Page number (shifted)       = 0x00000000
   Address of page table entry = 0x00001000
   Page table entry            = 0x00344003
       Frame number = 0x00344000
       V=1 (Page is valid)
       W=1 (Page is writable)
       R=0 (Page has not been referenced)
       D=0 (Page not dirty)
   Setting the referenced bit
   Physical address            = 0x00344468
   Modified page table entry   = 0x00344007
     (Page table entry was NOT modified)
   Translation completed with no exceptions
The value of the target word in physical memory was not changed.
It is...
  0x344468: 0x00000000
The page table has not been modified by this command.
>
```

The "Page Table Base Register" can be modified with the "setptbr" command. For example:

```
> setptbr
Enter the new value for the Page Table Base Register (PTBR) in hex: 2000
  PTBR = 0x00002000     ( decimal: 8192        )
>
```

The "Page Table Length Register" can be modified with the "setptlr" command. For example:

```
> setptlr
Enter the new value for the Page Table Length Register (PTLR) in hex: 10
  PTLR = 0x00000010     ( decimal: 16              HardwareFault )
>
```

Assembly Labels in the Emulator

A BLITZ assembly language program will define a number of "labels". Typically these are the targets of branch and call statements.

Consider this assembly code fragment:

```
getChLoop:                              ! loop
        cleari                          !   disable interrupts
        load    [r2],r3                 !   if (inBufferCount != 0)
        cmp     r3,0                    !   .
        bne     getChExit               !     then break
        seti                            !   enable interrupts
        jmp     getChLoop               ! end
getChExit:                              ! .
```

This code defines the labels "getChLoop" and "getChExit". When the linker determines where in memory this code will be placed, the linker will assign specific values to these labels. For example, the linker might place this code at location 0x000D10 in memory. Thus, these two labels will have these values after linking:

```
getChLoop     000D10
getChExit     000D28
```

The information about labels is not part of the program and is not used during the execution. Instead, the CPU uses relative and absolute byte addresses. Nonetheless, the labels and their values are placed in the executable file, along with the program bytes and the emulator reads this information in when it reads an executable file.

The emulator uses the label information when memory is disassembled. For example, if we disassemble the memory area containing these instructions, we will see the following. The disassembler inserts labels such as "getChLoop" and "getChExit" into the display, making it easier to understand.

```
> dis
Enter the beginning phyical address (in hex): 000d10
                  getChLoop:
000D10: 03000000      cleari
000D14: 6B320000      load    [r2+r0],r3
000D18: 81030000      sub     r3,0x0000,r0
000D1C: A300000C      bne     0x00000C        ! targetAddr = getChExit
000D20: 04000000      seti
000D24: A1FFFFEC      jmp     0xFFFFEC        ! targetAddr = getChLoop
                  getChExit:
000D28: 81330001      sub     r3,0x0001,r3    ! decimal: 1, ascii: ".."
000D2C: 6F320000      store   r3,[r2+r0]
...
>
```

In the BLITZ architecture, all jump, branch, and call instructions contain 24-bit relative offsets. In this example, the "jmp" instruction contains a relative offset of –20 (as a 24-bit hex value, 0xFFFFEC). The disassembler indicates that the target of the "jmp" is the instruction labeled "getChLoop".

To see all labels and their values, you can use the "labels" command. The list is printed twice. First, it is sorted alphabetically by the label name; second it is sorted by label value. (These lists are usually quite long; here we omit most of the output.)

```
> labels
Ordered alphabetically:
    Label                          Hex Value  (in decimal)
    ============================== =========  ============
    AddressException                0000001C            28
    AddressExceptionHandler         0000007C           124
    ...
    getChElse                       00000D4C          3404
    getChExit                       00000D28          3368
    getChLoop                       00000D10          3344
    getChar                         00001098          4248
    ...
    putStLoop                       00000E0C          3596
    ready                           00000C4C          3148
Ordered by value:
    Label                          Hex Value  (in decimal)
    ============================== =========  ============
    PowerOnReset                    00000000             0
    _entry                          00000000             0
    TimerInterrupt                  00000004             4
    DiskInterrupt                   00000008             8
    ...
    putChar2                        00000D74          3444
    putChLoop                       00000DA0          3488
    putChExit                       00000DB8          3512
    putChElse                       00000DDC          3548
    ...
    outBufferCount                  00006114         24852
    _Global_memoryArea              00006118         24856
    _Global_nextCharToUse           0000882C         34860
>
```

To find the value of a specific label, you can use the "find" command:

```
> find
Enter the first few characters of the label; all matching
labels will be printed: getCh
    Label                          Hex Value  (in decimal)
    ============================== =========  ============
    getChElse                       00000D4C          3404
    getChExit                       00000D28          3368
    getChLoop                       00000D10          3344
    getChar                         00001098          4248
    getChar2                        00000CE4          3300
>
```

If you know the value, but not the label, you may use the "find2" command:

```
> find2
Enter the value to find (in hex): d28
  00000D28   (decimal: 3368)   getChExit
>
```

You may also create a new label with the "add" command. For example:

```
> add
Enter the name of the new label: myLabel
Enter the value of the new label (in hex): d1c
Label "myLabel" has been added.
>
```

If we disassemble the same section of code as above, we will now see that the newly created label will be used just like any other label:

```
> dis
Enter the beginning phyical address (in hex): d10
                  getChLoop:
000D10: 03000000      cleari
000D14: 6B320000      load    [r2+r0],r3
000D18: 81030000      sub     r3,0x0000,r0
                  myLabel:
000D1C: A300000C      bne     0x00000C        ! targetAddr = getChExit
000D20: 04000000      seti
000D24: A1FFFFEC      jmp     0xFFFFEC        ! targetAddr = getChLoop
                  getChExit:
000D28: 81330001      sub     r3,0x0001,r3    ! decimal: 1, ascii: ".."
000D2C: 6F320000      store   r3,[r2+r0]
...
>
```

Of course the new label is added only to the tables maintained by the emulator. As soon as you quit the emulator (or execute the “reset” command), the newly added label will be gone.

In addition to printing decimal and ASCII equivalents, the disassembler also displays label information whenever it can. Sometimes this additional information is useful; other times it is meaningless and not useful. Consider this fragment of assembly code:

```
putChar:
            push    r14                     ! Function Entry:
            mov     r15,r14                 ! .  Setup the standard frame
            push    r13                     ! .
            set     RoutineDescriptor_putChar,r1
            push    r1                      ! .
            mov     0,r13                   ! .
            loadb   [r14+8],r1              ! Move the parameter into r1
            cmp     r1,'\n'                 ! IF (char != '\n')
            be      callputChar2            ! .
            cmp     r1,'\t'                 ! .  AND (char != '\t')
            be      callputChar2            ! .
            cmp     r1,' '                  ! .  AND (char < ' '
            bl      fixChar2                ! .
            cmp     r1,0x7e                 ! .     OR char > 0x7e)
            ble     callputChar2            ! .
fixChar2:                                   ! .
            mov     '?',r1                  !     char := '?'
callputChar2:                               ! END IF
            call    putChar2                ! Call putChar2
...
```

When disassembled, it prints like this. (Several of the long lines may wrap-around in this document.)

```
    > dis
    Enter the beginning phyical address (in hex): ff8
                        putChar:
    000FF8: 54EF0000        push    r14,[--r15]
    000FFC: 67EF0000        or      r15,r0,r14
    001000: 54DF0000        push    r13,[--r15]
    001004: C0100000        sethi   0x0000,r1       ! 0x00001064 = 4196
(RoutineDescriptor_putChar)
    001008: C1101064        setlo   0x1064,r1
    00100C: 541F0000        push    r1,[--r15]
    001010: 87D00000        or      r0,0x0000,r13
    001014: 8C1E0008        loadb   [r14+0x0008],r1 ! decimal: 8  (DiskInterrupt)
    001018: 8101000A        sub     r1,0x000A,r0    ! decimal: 10, ascii: ".."
    00101C: A2000020        be      0x000020        ! targetAddr = callputChar2
    001020: 81010009        sub     r1,0x0009,r0    ! decimal: 9, ascii: ".."
    001024: A2000018        be      0x000018        ! targetAddr = callputChar2
    001028: 81010020        sub     r1,0x0020,r0    ! decimal: 32, ascii: ". "
(PageInvalidException)
    00102C: A400000C        bl      0x00000C        ! targetAddr = fixChar2
    001030: 8101007E        sub     r1,0x007E,r0    ! decimal: 126, ascii: ".~"
    001034: A5000008        ble     0x000008        ! targetAddr = callputChar2
                        fixChar2:
    001038: 8710003F        or      r0,0x003F,r1    ! decimal: 63, ascii: ".?"
                        callputChar2:
    00103C: A0FFFD38        call    0xFFFD38        ! targetAddr = putChar2
    ...
    >
```

First consider the assembly source code instruction:

```
    cmp     r1,'\n'                 ! IF (char != '\n')
```

which is disassembled as:

```
    sub     r1,0x000A,r0    ! decimal: 10, ascii: ".."
```

Recall that the compare instruction “cmp” is a synthetic instruction; it is actually assembled as a subtract instruction, with the result stored into “r0” (i.e., the result is discarded). The disassembler prints the instruction as it actually is.

When a literal value is included in an instruction, the disassembler will print it in hex (e.g., “0x000A”). In the comment area, the disassembler also prints this value in decimal and in ASCII. (The literal is two bytes long, so the disassembler prints two ASCII characters, enclosed in quotes. Since the neither “00” nor “0A” are considered printable characters, the disassembler prints two dots between the quotes.)

Next consider the assembly source code instruction:

```
    cmp     r1,' '                  ! .  AND (char < ' '
```

which is disassembled as:

```
    sub     r1,0x0020,r0    ! decimal: 32, ascii: ". "  (PageInvalidException)
```

Of the two bytes in the literal value (0x00 and 0x20), the first is not a printable ASCII character code and the second is the ASCII “space” character; between the quotes we see first a dot, then a space.

By coincidence, the literal value in this instruction also happens to be the value of a label named "PageInvalidException". The disassembler also includes this information, although it is not relevant or helpful in this case.

When the disassembler encounters a "sethi" instruction followed by a "setlo" instruction, it assumes they are the result of a synthetic "set" instruction. For example, consider the assembly source instruction:

```
    set     RoutineDescriptor_putChar,r1
```

which is disassembled as:

```
    001004: C0100000       sethi   0x0000,r1       ! 0x00001064 = 4196
(RoutineDescriptor_putChar)
    001008: C1101064       setlo   0x1064,r1
```

The disassembler puts together the two literals (0x0000 and 0x1064) in the two instructions to get a combined 32-bit value. This value is printed in decimal (4196) and as a label (if a label with this value exists). In this case, the label is meaningful and helpful, while the decimal value is not.