The University of Texas at Austin
Department of Electrical and Computer Engineering

Real-Time Systems / Real-Time Operating Systems

EE445M/EE380L.6, Spring 2016

Final Exam Solutions

Date: May 12, 2016

UT EID: _______________

Printed Name: ____
Last, First

Your signature is your promise that you have not cheated and will not cheat on this exam, nor will you help others to cheat on this exam:

Signature: ____

Instructions:

- Open book and open notes.
- No calculators or any electronic devices (turn cell phones off).
- Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space (boxes) provided.
- *Anything outside the boxes will be ignored in grading.*
- For all questions, unless otherwise stated, find the most efficient (time, resources) solution.

Problem 1	10	
Problem 2	10	
Problem 3	15	
Problem 4	10	
Problem 5	20	
Problem 6	25	
Problem 7	10	
Total	100	

*Name:*______________________________

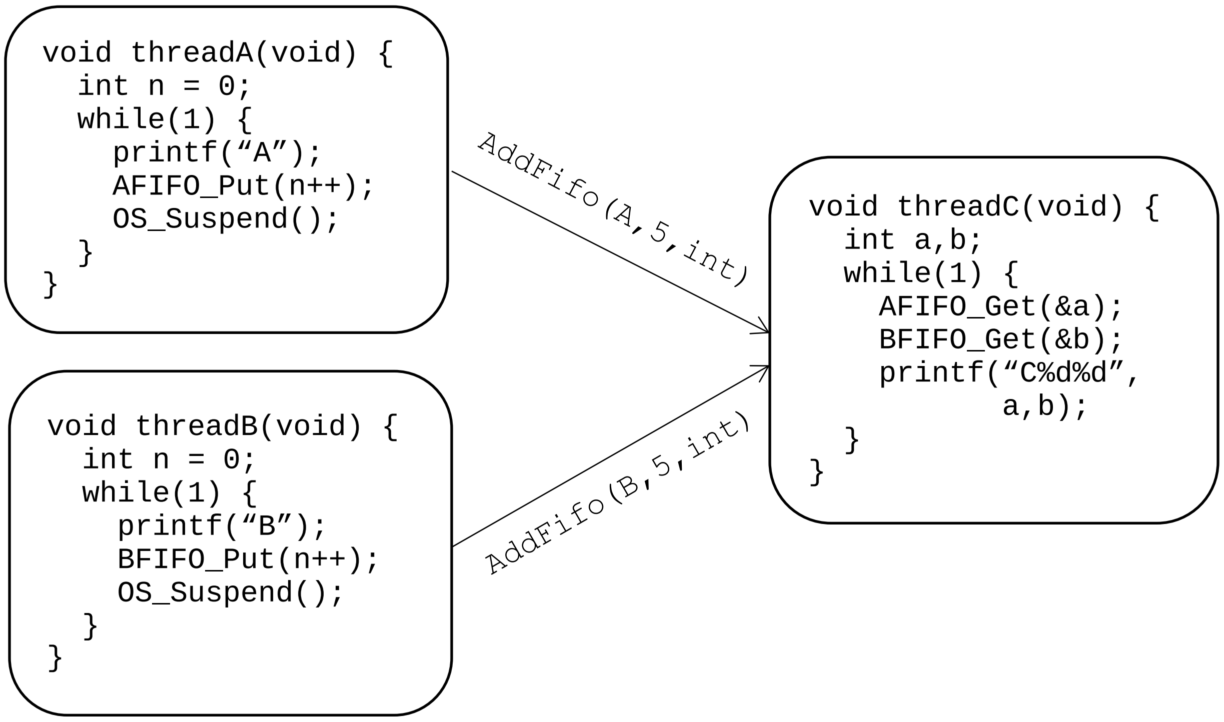
Problem 1 (10 points): Synchronization

Given the macro-based basic FIFO implementation below, insert mutexes to make the FIFO fully thread-safe, i.e. resolve all critical sections against multiple concurrent foreground threads calling *Get*() and *Put*() concurrently. Also insert counting semaphores for `roomLeft` and `dataAvailable` to wait and block calling threads until there is room in the FIFO on *Put*() and data available on *Get*():

```
#define AddFifo(NAME,SIZE,TYPE)                                 \

unsigned long volatile PutI ## NAME;                            \
unsigned long volatile GetI ## NAME;                            \

                           sema_t  mutex, roomLeft, dataAvailable; \

TYPE static Fifo ## NAME [SIZE];                                \
void NAME ## Fifo_Init(void){                                   \
  PutI ## NAME= GetI ## NAME = 0;                               \

                           OS_InitSemaphore(&mutex, 1);           \
                           OS_InitSemaphore(&roomLeft, SIZE);     \
                           OS_InitSemaphore(&dataAvailable, 0);   \

}                                                               \
void NAME ## Fifo_Put (TYPE data){                              \

                           OS_Wait(&roomLeft);                    \
                           OS_bWait(&mutex);                      \

  Fifo ## NAME[ PutI ## NAME &(SIZE-1)] = data;                 \
  PutI ## NAME ## ++;                                           \

                           OS_bSignal(&mutex);                    \
                           OS_Signal(&dataAvailable);             \

}                                                               \
void NAME ## Fifo_Get (TYPE *datapt){                           \

                           OS_Wait(&dataAvailable);               \
                           OS_bWait(&mutex);                      \

  *datapt = Fifo ## NAME[ GetI ## NAME &(SIZE-1)];              \
  GetI ## NAME ## ++;                                           \

                           OS_bSignal(&mutex);                    \
                           OS_Signal(&roomLeft);                  \
}
```

Problem 2 (10 points): Scheduling

Given the following KPN-like application code running on top of your OS and using blocking FIFOs as described and implemented in Problem 1. All FIFOs have a size of 5 integers. Assume that *threadA* is the first thread to be launched by the OS:



```
void threadA(void) {
  int n = 0;
  while(1) {
    printf("A");
    AFIFO_Put(n++);
    OS_Suspend();
  }
}
```

```
void threadB(void) {
  int n = 0;
  while(1) {
    printf("B");
    BFIFO_Put(n++);
    OS_Suspend();
  }
}
```

```
void threadC(void) {
  int a,b;
  while(1) {
    AFIFO_Get(&a);
    BFIFO_Get(&b);
    printf("C%d%d",
           a,b);
  }
}
```

a) What will be the output printed by the program if it is running under a non-preemptive, i.e. cooperative round-robin scheduler? Will any of the FIFOs ever be full?

Output (round-robin order A->B->C->A->…): ABC00ABC11ABC22…
Ppossibly also (round-robin order A->C->B->A->…): ABAC00BAC11BAC22…

FIFOs will never fill up.

b) What will be the output of the program under a strict priority scheduler with threads *A*, *B* and *C* having high, medium and low priority, respectively? Will any of the FIFOs ever be full?

Output: AAAAAABBBBBBBABC00ABC11ABC22…
Or (if OS_Signal() doesn't immediately context switch): AAAAAABBBBBBC00C11C22C33C44ABC55…

FIFOs do fill up. In the first (strict priority) case, they fill up in beginning and remain full.
In the second case, they fill up in the beginning, then get emptied, then fill up again, and so on.

*Name:*______________________________

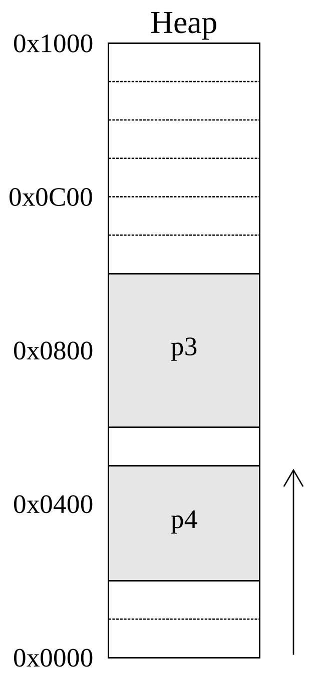
Problem 3 (15 points): Memory Management

Consider a 4kB heap and the following sequence of heap allocations and de-allocations triggered, for example, by different programs being loaded/launched and exiting:

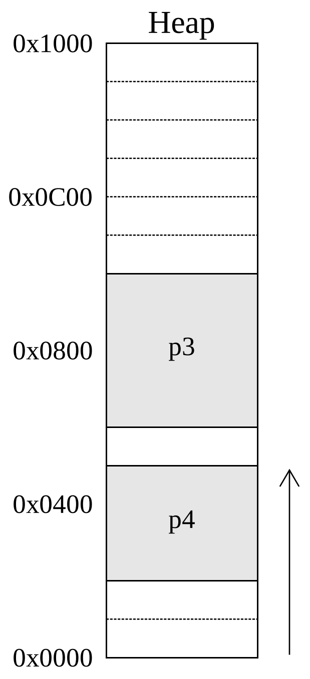
Program 1 launches:	`p1 = Heap_Malloc(512);`
Program 2 launches:	`p2 = Heap_Malloc(1024);`
Program 3 launches:	`p3 = Heap_Malloc(1024);`
Program 2 exits:	`Heap_Free(p2);`
Program 4 launches:	`p4 = Heap_Malloc(768);`
Program 1 exits:	`Heap_Free(p1);`

For each of the different heap allocation strategies below, show the state of the heap at the end of this sequence. Mark the memory regions allocated to each program that is loaded into memory at this point. Is the heap fragmented? What is the amount of free space and what is the largest block/program size that can be allocated/loaded then? Assume that the heap manager does not require any overhead for extra meta-data, and that the heap is allocated from bottom to top, i.e. a block always ends up being placed at the bottom of its chosen free space region.

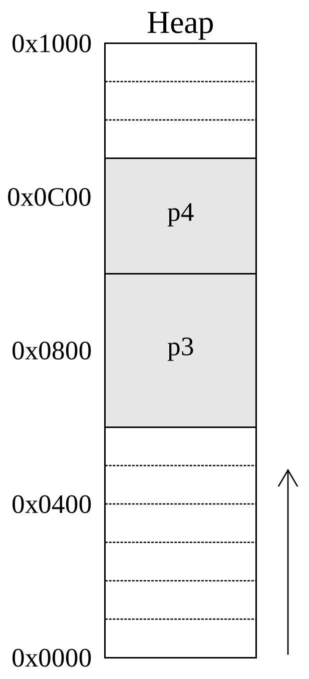
a) First fit



b) Best fit



c) Worst fit



Fragmented: Yes *Free space: 9 * 256 = 2304 bytes* *Max. alloc: 6 * 256 = 1.5kB*	*Behaves the same as First Fit:* *Fragmented: Yes* *Free space: 9 * 256 = 2304 bytes* *Max. alloc: 6 * 256 = 1.5kB*	*Fragmented: Yes* *Free space: 9 * 256 = 2304 bytes* *Max. alloc: 6 * 256 = 1.5kB*

Problem 4 (10 points): Supervisor Call

a) Given below is the code template for an *SVC_Handler* given in class. Complete the code fragment to realize a supervisor call to *OS_AddThread* whenever a `SVC #42` instruction is executed.

```
SVC_Handler

    LDR  R12,[SP,#24]       ; Get return address from stack

    LDRH R12,[R12,#-2]      ; Load SVC instruction (2 bytes)

    BIC  R12,#0xFF00        ; Extract ID in R12

    LDM  SP,{R0-R3}         ; Get any parameters

    CMP  R12,#42            ; check for correct ID
    BNE  EndSVC             ; do nothing if not
    PUSH {LR}               ; save link register
    BL   OS_AddThread       ; call actual OS routine
    POP  {LR}               ; restore link register

EndSVC

    STR  R0,[SP]            ; Store return value

    BX   LR                 ; Return from exception
```

b) In earlier lectures we had discussed that invocation of OS kernel routines via supervisor calls (SVCs) will also be necessary when running an OS that uses the Process Stack Pointer (PSP) for all user code. Would your handler above also work in such a setup? If not, show the necessary modifications in the code above for the handler to work in a PSP setup.

Modifications are needed. Return address, return value, function parameters are all on the PSP.

Replace start of handler with:

```
        MRS     R12,PSP           ; Get PSP
        LDM     R12,{R0-R3}       ; Get parameters from process stack
        LDR     R12,[R12,#24]     ; Get return address from process stack
        LDRH    R12,[R12,#-2]     ; Load SVC instruction
        BIC     R12,#0xFF00       ; Extract ID
```

And end of handler with:

```
        MRS     R12,PSP           ; Get PSP
        STR     R0,[R12]          ; Store return value on process stack
        BX      LR
```

Problem 5 (20 points): Networking

a) Assume a CAN 2.0A bus running at a baud rate of 1Mbit/s (=1,000,000 bit/s) and using 11-bit IDs, what is the maximum bandwidth achievable for CAN data transfers? You don't have to compute the actual result, just showing the expression is ok.

$$\text{Bandiwidth} = \frac{\text{Data bits}}{\text{Frame delay}} = \frac{\text{64 bits}}{\text{11+36+64 bits}} * \text{1,000,000 bit/s} = \frac{64}{111}\ \text{Mbit/s}$$

b) What is the bandwidth when (continuously) transmitting 32-bit data values of 0x00FF00FF? Hint: Don't forget about bit stuffing. Again, just the expression not final number is ok.

Bit stuffing inserts one extra bit every five consecutive equal bits, so 4 additional stuff bits here for the data. Bit stuffing also happens for other parts of the frame depending on the bit patterns, e.g. for the ID, but assumed to be no stuffing needed elsewhere here:

$$\text{Bandwidth} = \frac{32}{11+36+4+32}\ \text{Mbit/s} = \frac{32}{83}\ \text{Mbit/s-}$$

c) Assuming a CAN network with 3 nodes. At time 0, node 0 wants to send a message with ID 42 to node 1, node 1 wants to send a message with ID 14 to node 2 and node 2 wants to broadcast a message with ID 4 to nodes 0 and 1. At what time does each of the three messages reach their destination(s)? Just show the result as a function of the frame delay t_f (= time to complete a single message transfer).

Highest priority message 4, broadcast 2 -> 0,1 finishes at time t_f
Medium priority message 14, 1->2 finishes at $2*t_f$
Lowest priority message 42, 0->1 finishes at $3*t_f$

d) Now consider an SPI bus running at a clock rate of 8MHz (=8,000,000 bit/s). What is the maximum achievable bandwidth for (continuously) reading single data blocks of 512 bytes assuming zero command-response delay (NCR=0) and an immediate data start (i.e. zero data packet delay)? How does that compare to the SD card bandwidth you measured in Lab 4?

$$\text{Bandwidth} = \frac{\text{512 byte data * 8 Mbit/s}}{\text{6byte(command)+1byte(response)+515byte(packet)}} = \frac{256}{261}\ \text{MB/s} \approx 1\ \text{MB/s}$$

Should be about 4x-8x higher than what was seen in Lab 4 (depending on the SD card).

Problem 6 (25 points): Filesystem

The FAT16 filesystem standard uses a file allocation table (FAT) scheme with 16-bit FAT entries. The first two blocks on disk and corresponding FAT entries are reserved to store the boot sector, the root directory and one or more redundant copies of the FAT itself. FAT entries of zero indicate an empty block, while any value larger than 0xFFF8 is used as end-of-file marker. Each directory entry uses 32 bytes. Given the following datastructures for a FAT16-compliant filesystem (directory and FAT):

```
// directory w/ 32 byte entries
struct dir_entry {
  char   name[8];  // file name
  char   ext[3];   // extension
  uint8  attr[11]; // attributes
  uint16 date;     // modified..
  uint16 time;     //  ..date/time
  uint16 start;    // start block
  uint32 size;     // size in byte
} dir[DIR_SIZE];

uint16 fat[FAT_SIZE]; // FAT
```

	FAT
0	
1	
2	0x0004
3	0x0000
4	0x000C
5	0x0000
6	0xFFFF
7	0x0000
8	0x0000
9	0x000A
10	0x0006
11	0x0000
12	0x000D
13	0x000E
14	0xFFFF
15	0x0009

	Disk
0	
1	
2	A0*
3	
4	A1
5	
6	B3
7	
8	
9	B1
10	B2
11	
12	A2
13	A3
14	A4
15	B0*

a) For the specific instance of a FAT with 16 entries as shown in the figure above, how many files are there on disk and what is the size of each of these files assuming 512 byte disk blocks. Mark the blocks belonging to each file and each file's starting block in the disk layout on the right. How much free space is there left on the disk?

*2 files, file A and B as marked above. Starting blocks A0 and B0 as indicated with *.*

*Free space: 5 blocks * 512 bytes/block= 2.5kB*

b) What is the maximum file size supported by FAT16, and what part of the FAT16 datastructure determines the maximum file size?

Maximum file size determined by 'size' directory entry: 2^{32} byte = 4 GiB
Upper bound on file size also determined by maximum disk size, see below.

c) Assuming 512 byte disk blocks, what is the largest disk size supported by FAT16, and how many entries does the FAT maximally have? What determines the maximum disk size? How large does the FAT need to be for a disk with 128kB?

Largest disk size & why?	*16-bit FAT entries, so maximally 2^{16} blocks = 65536 blocks * 512 byte = 32MiB* *In reality, FAT entries must be <= 0xFFF7, so disk size limit is 8 blocks (4kB) less.*
Max. `FAT_SIZE`	*One entry per disk block: 2^{16} = 65,536 (-8)*
`FAT_SIZE` for 128kB disk	*128 * 1024 / 512 = 256 entries*

d) To support larger disk sizes, the FAT16 standard in reality uses clustering, where the disk is partitioned into fixed-sized clusters and each FAT entry refers to a cluster of contiguous blocks/sectors on disk. What is the cluster size needed for a disk with 2^{27} bytes? What is the disadvantage of clustering?

$2^{27} / 2^{16} = 2^{11}$ bytes/cluster = 2kB clusters

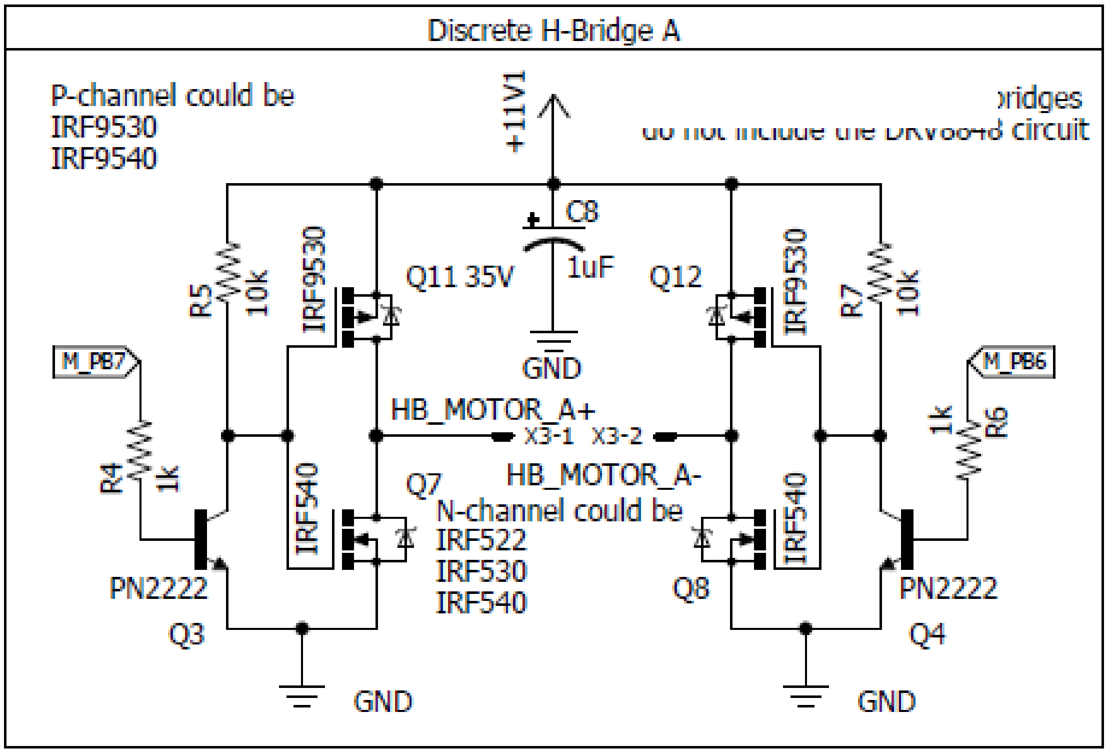
Increases internal fragmentation.

e) Assuming a FAT16-formatted disk with 4kB cluster size and sectors of 512 bytes, show the code of a filesystem routine that reads the *n*-th byte from the file *handle* pointing to its directory entry. You can assume that the `dir` and `fat` datastructures have already been read and populated from disk, where `start` directory fields points to a file's first cluster.

```
// Return 0 if successful, 1 on failure
int FAT16_ReadByte(struct dir *handle, long n, char* pt)
{
  char buf[512];


  uint16 cluster;
  uint32 block;

  if(n >= handle->size) return 1;   // beyond end of file?

  cluster = handle->start;   // get index of first cluster of file
  while(n >= 4096) {         // go to cluster that contains byte n
    n -= 4096;
    cluster = fat[cluster];  // walk along linked list
  }

  block = cluster * 8;       // starting block of cluster on disk
  block += n / 512;          // block within cluster that contains byte n


  if(eDisk_ReadBlock(buf,   block  ) return 1;


  *pt = buf[n % 512];        // get correct byte out of block



  return 0;
}
```

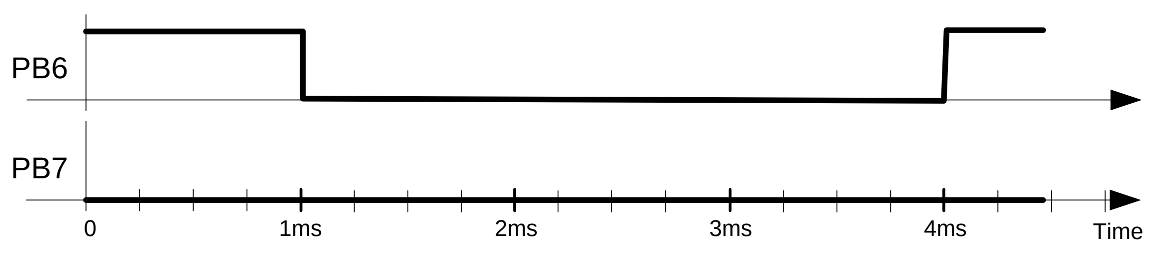
Problem 7 (10 points): Motor Driver

Shown below is the schematic of the H-bridge on our motor board. Assume that the motor runs in "forward" direction if current is flowing from A+ to A-:



For each of the following waveforms (with 4ms period), describe precisely what will happen if they are applied to the PB6 and PB7 inputs? Is the motor running? If so, in what direction and at what speed? Etc.

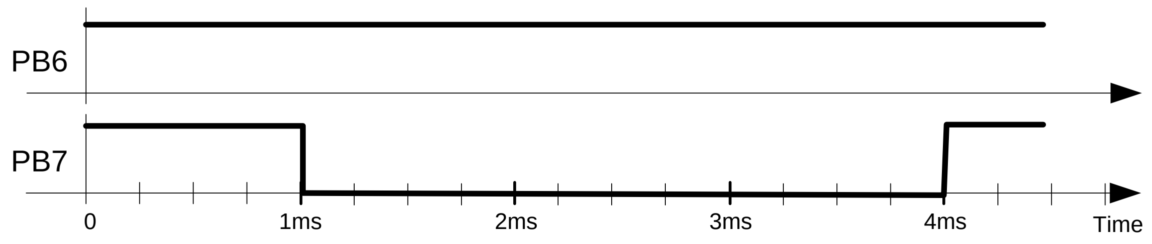
a)



Motor is running in "backward" direction at 25% speed.

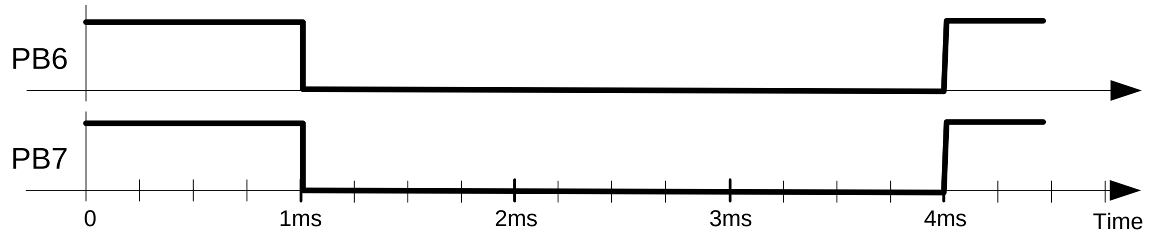
Name:_______________________________

b)



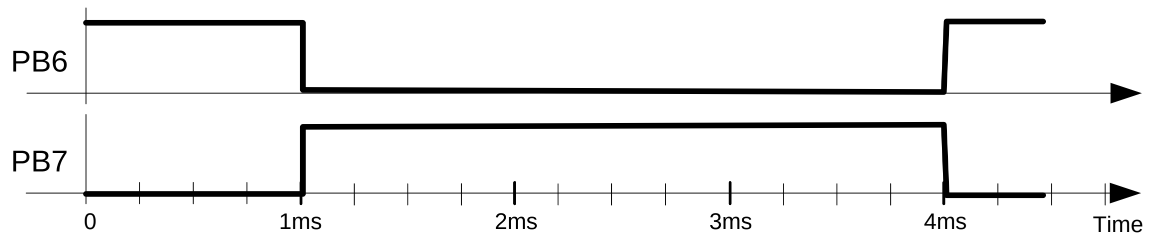
Motor is running in "backward" direction at 75% speed.

c)



Motor is not running.

d)



Power is oscillating between 25% backward and 75% forward, either no movement at all (due to inertia of motor) or slight, jerky forward movement.