

ECEN 2120: Exam #1  
Autumn 2009, Prof. Aaron Bradley

Name: \_\_\_\_\_

*I will uphold the CU Boulder honor code.*

Signature: \_\_\_\_\_

**Instructions:** Implement the following two (2) specified functions in MC68K assembly, adhering to the C/assembly-interface standards that we've used thus far. Each problem is worth half the grade. I will award partial credit. Comments will certainly help the grading process (indicating, for example, what you intended to do if the implementation itself is slightly wrong) and might yield more partial credit. Whatever you do, don't panic: take a deep breath, and give it your best shot.

The second physical page of this document is almost completely blank (having just a place to write your name) and is intended to provide space to write your solutions.

## 1 !yaD sdrawkcaB s'tI

```
/* In:  source - a C-string (null-terminated)
 *      dest - the destination to which to write the reverse of source
 *      n - the size of dest
 * Out:  0 - if successful
 *      -1 - if the size of dest (n) is too small to hold the reverse
 * Writes the reverse of the string at source into dest, unless the
 * size of dest, given by n, is too small to hold the whole string.
 * At the ending (if no error occurred), dest should be a C-string.
 * For example, if I declared
 *  NAME  dc.b  'ecen2120',0    ; the C-string "ecen2120"
 *  RNAME ds.b  256            ; plenty of space
 * then after calling reverse_string, examining memory at RNAME would
 * reveal
 *  '0212nece',0
 * (as well as 247 subsequent garbage bytes) that is, the C-string
 * "0212nece".
 *
 * Recommended steps:
 * 1. locate the null-terminator (ASCII value 0) in source
 * 2. if it's at position n or beyond, return -1
 * 3. otherwise, write the reverse of the string into dest
 * 4. write a null-terminator (value 0) at the end of what was written
 *    into dest
 * 5. return 0
 */
short reverse_string(char * source, char * dest, unsigned short n);
```

## 2 Monocycle

```
/* In: table - a table of shorts, each in the range [0, n] (i.e., at
 *      least 0 and at most n)
 *      n - the number of entries in table, and the upper bound on
 *          their values; assume that n > 0
 * Out: 0 - if no cycle
 *      1 - if there is a cycle
 * Decides whether table has a cycle. The definition of a cycle is
 * best explained by example. Consider two tables:
 * table1 = { 3, 5, 1, 2, 1 },
 * table2 = { 3, 2, 1, 2, 5 }.
 * table1 does not have a cycle: start with index 0, so
 * table1[0] = 3, and
 * table1[3] = 2, and
 * table1[2] = 1, and
 * table1[1] = 5, terminating the sequence
 * To determine that it doesn't have a cycle, I used each entry as the
 * next index (where 0 is my first index by definition) until I found
 * the value 5 (the length of table1). In contrast, table2 has a
 * cycle:
 * table2[0] = 3, and
 * table2[3] = 2, and
 * table2[2] = 1, and
 * table2[1] = 2, and
 * table2[2] = 1, indicating a cycle
 * To decide that table2 has a cycle, I only need to follow the
 * indexes 5 times (n times in general) without seeing 5 (n in
 * general) to conclude that there definitely must be a cycle. (There
 * are other ways of deciding whether there is a cycle, but this one
 * is easy to implement.) Additional examples:
 * { 0, 1, 2, 3 } has a cycle
 * { 1, 2, 3 } does not have a cycle
 * { 2, 1, 0 } has a cycle
 * { 2, 3, 0 } has a cycle
 * { 2, 3, 1 } does not have a cycle
 * Because n > 0, there will always be at least one entry.
 *
 * Recommended steps:
 * 1. starting with index 0, follow the entries through the table,
 *    using each successive entry as the index to find the next entry
 * 2. if the entries are followed n times without encountering the
 *    value n, return 1 (a cycle)
 * 3. otherwise (if value n is encountered), return 0 (no cycle)
 */
short cycle(unsigned short * table, unsigned short n);
```

Name: \_\_\_\_\_

