

1 Introduction

In this lab, you will write a program that runs one of two memory tests based on the input from a user interface. Because this lab is more complex, it will be designed in two hierarchical steps. In this case, we will design bottom up as we have not yet covered the BasicIO routines required for the user interface in lecture.

Part One: design the memory test subroutines, which will be called from a simple calling routine.

Part Two: Design a calling routine that implements a user interface using the BasicIO routines.

Part One - You will write a program that will run a ROM test and a RAM test. The ROM test will use a checksum as described in Appendix A of this description. The RAM test is a simple alternating 1 and 0 test as described in Appendix B.

Part Two – You will use the BasicIO routines to design a user interface as described in the Requirements.

This lab tests your ability, to create flow diagrams, design subroutines with parameters, and use the BasicIO module.

The challenging aspects of the lab are:

- It is a complex program. Therefore, it requires careful design using flow diagrams.
- The user-interface and input parsing requires a good understanding of the Basic I/O routines.

2 Requirements – Part One

The program must meet the following requirements:

1. It must consist of the main program, a checksum subroutine, *ChkSum()*, and a RAM test subroutine, *RAMTest()*. The main program passes the specified memory block parameters to each subroutine and responds as specified to the test results. This part results in a program that runs one time and stops – clearly it is not a finished product because it does not run in an endless loop.
2. The main program must do the following:
 - a. Call the RAM test subroutine to have it test the address block, \$3000-\$37FF.
 - b. If a RAM error occurred, turn on LED13 (PORTA0) and stop by implementing a trap.
 - c. If there was no RAM error, call the checksum routine to have it test block \$C000-\$FFFF. Store the returned checksum value in a global variable, *CSResult*
 - d. If the calculated checksum was not equal to \$90D2, turn on LED12 (PORTA1) and stop by implementing a trap.
 - e. If the checksum was correct, turn on LED11 (PORTA2) and stop by implementing a trap.
3. *ChkSum()* - The checksum subroutine calculates the checksum as described in Appendix A. It must be a general purpose subroutine with the following parameter specification:

Parameters. The checksum subroutine must have the starting address of the block passed in ACCD and ending address of the block passed on the stack. The calculated checksum must be returned in ACCD.
4. *RAMTest()* - The RAM test subroutine must perform the RAM test as described in Appendix B. It must be a general purpose subroutine with the following parameter specification:

Parameters. The RAM test subroutine must have the starting address of the block passed in ACCD and ending address of the block passed on the stack. If there is an error, the address where the error occurred should be returned in ACCD with the C flag set. If there is no error, it should return with the C flag cleared.

There are five data objects required for this program – A variable that contains the resulting checksum, *CSResult*, two stored constants for the starting, *ROMStartAddr*, and ending, *ROMEndAddr*, addresses of the ROM block to be tested, and two stored constants for the starting, *RAMStartAddr*, and ending, *RAMEndAddr*, addresses of the RAM block to be tested. These are the values to be passed to the subroutines.

Note that the starting and ending addresses are stored constants so they must be created using the *dc* directive.

3 Testing – Part One

1. To test your ROM test, verify that checksums agree with those listed below. The two blocks shown will test both even and odd block sizes along with checking for the terminal count bug.

Test Block	Checksum
\$C000-\$FFFF	\$90D2
\$C001-\$FFFF	\$9086

The second test will cause a checksum error so the LED should turn on. The actual checksum should be stored in *CSResult*. To change the starting address for the second block, use D-Bug12 to change the memory location in which starting address constant is stored.

2. To verify that your RAM test accurately tests even and odd blocks perform the following tests:
 - Using the D-Bug12's block fill instruction, *BF*, fill \$3000-\$300F with \$00. With your program, test the block \$3001-\$3002. The test should pass and both \$3001 and \$3002 should be filled with an \$AA and \$55 or a \$55 and \$AA. All other bytes from \$3000-\$300f must remain \$00.
 - Repeat test a. with the odd sized block, \$3001-\$3003.
 - Test the even sized block, \$8000-\$8001 and the odd sized block \$8000-\$8002. These two tests should fail because they are not addresses that point to RAM.
 - Note there is no easy way to test your program with an ending address of \$FFFF or a starting address of \$0000. Carefully analyze your program to verify that it would work at those two addresses.

Again, to change the starting address and ending addresses for the tests, use D-Bug12 to change the memory location in which starting address constant is stored.

4 Deliverables – Part One

The source code for your program must be sent via email by midnight on the due date. There is no write-up for this part of the lab.

Due Date: Source-Tuesday, Feb 5, 2008

5 Requirements – Part Two

In this part of the lab, you will replace the main program from Part One with a program that implements a user interface to determine the memory blocks to be tested.

5.1 User interface requirements:

- A. On a new line, prompt the user to proceed with a test. The prompt must indicate the options **RAM** or **ROM** for the *Ram Test* or the *ROM Checksum Test*.
- All other entry combinations do nothing but redisplay the prompt on a new line.
 - The user response is not processed until a **CR** is received or >4 characters are received. This gives the user an opportunity to change the selection using a backspace.
 - All user input for this program is NOT case sensitive.
- B. If the user enters **RAM**, prompt the user for test block information.
- The program must output a message that indicates what test is being run and prompts for the starting and ending addresses (inclusive) of the block to be tested. You may use two separate prompts or a single prompt to receive both addresses on one line.
- Note: While the RAM test must be designed to accept and work with any start and end address be careful not to test over your code, variables, or the stack.
- Again, user inputs are not processed until a **CR** is received. Backspace must work.
 - If a **CR** is entered first for either address, quit the test and go back to A. to redisplay the opening prompt.
 - When a user-input error occurs, an explanatory message is displayed and, on a new line, the program prompts for new addresses. The following errors must be detected: illegal hex characters, hex value too large, and the ending address is lower than the starting address.
 - After the test is completed without an error, display:
RAM Test Passed↵
on a new line, or if an error did occur, display:
RAM Test Failed at XXXX↵
on a new line where, **XXXX** is the address where the test failed and ↵ is a carriage return.
Then go back to A. and display the opening prompt.
- C. If **ROM** is received, prompt the user for test block information.
- The program must output a message that indicates what test is being run and prompts for the starting and ending addresses (inclusive) of the block to be tested. You may use two separate prompts or a single prompt to receive both addresses on one line.
 - Note, the ROM test must be designed to accept and work with any start and end address.
 - Again, user inputs are not processed until a **CR** is received. Backspace must work.
 - If a **CR** is entered first for either address, quit the test and go back to A.
 - When a user-input error occurs, an explanatory message is displayed and, on a new line, the program prompts for new addresses. The following errors must be detected: illegal hex characters, hex value too large, and the ending address is lower than the starting address.
 - When the checksum calculation is complete, the following is displayed on a new line:
CS: XXXX↵
where, **XXXX** is the resulting checksum and ↵ is a carriage return.
 - After displaying the checksum, the program returns to part A.

6 Testing – Part Two

1. Test the ROM and RAM as described in Part One. Only this time you will enter the addresses through the user interface.

ETec374, Lab 4 – Memory Tests with User Interface

2. To verify that your user interface catches user errors:

- Check for bad character input
- Check for illegal address entry
- Check for balanced stack after the program is run.

7 Write-up

The source code for your program must be sent via email by midnight on the due date. The write-up must be turned in before 5:00pm on the due date and include the following material:

Introduction

Program Description - Including a flow diagram.

Program Listing (hardcopy)

Comments and Conclusions

Due Dates: Source-Feb 15, 2008; Write-up-Feb 27, 2007

Appendix A – ROM Tests

In this lab, you will write a program that tests a ROM memory block by verifying its contents. If a ROM test is performed when the MCU is first powered-up, one can avoid more catastrophic failures during normal operation.

There are two common ways to test ROM contents, the checksum (or sumcheck) and the Cyclical Redundancy Check (CRC). The checksum is the simplest to perform while the CRC is a better test due to a more randomized result. In this lab, you will write a program to calculate the checksum of a memory block.

A checksum is defined as the 16-bit sum of each byte in the block. That is, an unsigned 8-bit addition of each byte with an unsigned 16-bit result. For example:

If the contents of a 5-byte EPROM block are: \$1F, \$FE, \$13, \$05, \$CD

The resulting 16-bit checksum of the 5-byte block is: \$0202

Carries out of the 16-bit result are ignored.

The resulting checksum is a number that can be used to identify the contents of memory. Of course, a 16-bit number can not be unique for every combination of the memory contents. Even if the number is not unique, it is rare for an error in memory that results in the same checksum value.

For example, if second location in the EPROM block above failed to hold its charge, the \$FE would change to \$FF. This failure would result in a checksum of \$0203 so the error would be detected. If however, this error was coupled with an error that caused the third location to go to \$12, the checksum would be the same and the errors would not be detected. This type of error is rare. However, in these cases, a CRC would detect both errors and would be a better, but more complex, solution.

Appendix B – RAM Tests

A RAM test should always be performed early on in the power-up sequence of a microcomputer system. By detecting a bad RAM device early, errors that are more serious can be avoided during normal operation. There are many algorithms for testing RAM. Some are very simple – and fast – tests that will detect most hard failures of memory data and – if lucky – some soft failures and address errors. More complex tests will test both data and address errors along with subtle soft failures due to the actual layout of the memory IC.

Power-up RAM test are normally fast and simple tests. This is the type of test you will write for this lab. This simple test works by filling the memory block to be tested with alternating \$55 and \$AA. After being filled, the block is then read to verify the contents. i.e. each byte still contains the \$55 or \$AA. The test is then repeated with the opposite sequence \$AA and \$55. In this way, it can be confirmed that every bit in the RAM block can hold a '0' and a '1'.

Example: Test the block \$3000-\$3003

The first time through the test fills the bytes, in order, with: \$55, \$AA, \$55, \$AA

It then verifies that the contents are correct.

Then it fills the bytes, in order, with: \$AA, \$55, \$AA, \$55

And, again, it verifies that the contents are correct.