Software-based memory testing

By Michael Barr
Technical Editor
Embedded Systems Programming

One piece of software that nearly every embedded developer must write at some point in his career is a memory test. Often, once the prototype hardware is ready, the board's designer would like some reassurance that she has wired the address and data lines correctly, and that the various memory chips are working properly. Even if that's not the case, it is desirable to test any onboard RAM at least as often as the system is reset. It is up to the embedded software developer, then, to figure out what can go wrong and design a suite of tests that will uncover potential problems.

At first glance, writing a memory test may seem like a fairly simple endeavour. However, as you look at the problem more closely you will realise that it can be difficult to detect subtle memory problems with a simple test. In fact, as a result of programmer naiveté*, many embedded systems include memory tests that would detect only the most catastrophic memory failures. Perhaps unbelievably, some of these may not even notice that the memory chips have been removed from the board!

The purpose of a memory test is to confirm that each storage location in a memory device is working. In other words, if you store the value 50 at a particular address, you expect to find that value stored there until another value is written to that same address. The basic idea behind any memory test, then, is to write some set of data to each address in the memory device and verify the data by reading it back. If all the values read back are the same as those that were written, then the memory device is said to pass the test. As you will see, it is only through careful selection of the set of data values that you can be sure that a passing result is meaningful.

Of course, a memory test like the one just described is necessarily destructive. In the process of testing the memory, you must overwrite its prior contents. Since it is usually impractical to overwrite the contents of nonvolatile memories, the tests described in this article are generally used only for RAM testing. However, if the contents of a non-volatile memory device, like flash, are unimportant-as they are during the product development stage-these same algorithms can be used to test those devices as well.

Table 1: Consecutive data values for the walking 1's test

00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000

Common memory problems

Before implementing any of the possible test algorithms, you should be familiar with the types of memory problems that are likely to occur. One common misconception among software engineers is that most memory problems occur within the chips themselves. Though a major issue at one time (a few decades ago), problems of this type are increasingly rare. These days, the manufacturers of memory devices perform a variety of post-production tests on each batch of chips. If there is a problem with a particular batch, it is extremely unlikely that one of the bad chips will make its way into your system.

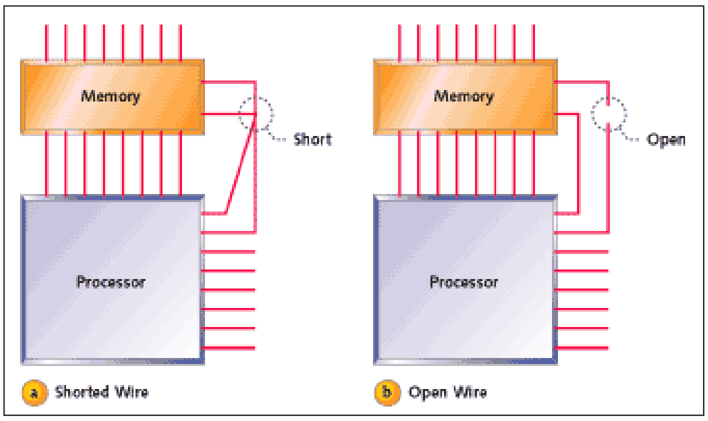

Figure 1: Possible Wiring Problems

The one type of memory chip problem you could encounter is a catastrophic failure. This is usually caused by some sort of physical or electrical damage to the chip after manufacture. Catastrophic failures are uncommon and usually affect large portions of the chip. Since a large area is affected, it is reasonable to assume that catastrophic failure will be detected by any decent test algorithm.

In my experience, the most common source of actual

Listing 1: Data bus test

```
typedef unsigned char datum; /* Set the data
bus width to 8 bits. */
datum
memTestDataBus(volatile datum * address)
{
datum pattern;
/*
* Perform a walking 1's test at the given
address.
*/
for (pattern = 1; pattern != 0; pattern
<
<
= 1)
{
/*
* Write the test pattern.
*/
*address = pattern;
/*
* Read it back (immediately is okay for this
test).
*/
if (*address != pattern)
{
return (pattern);
}
}
return (0);
} /* memTestDataBus() */
```

memory problems is the circuit board. Typical circuit board problems are problems with the wiring between the processor and memory device, missing memory chips, and improperly inserted memory chips. These are the problems that a good memory test algorithm should be able to detect. Such a test should also be able to detect catastrophic memory failures without specifically looking for them. So, let's discuss the circuit board problems in more detail.

Electrical wiring problems

An electrical wiring problem could be caused by an error in design or production of the board or as the result of damage received after manufacture. Each of the wires that connect the memory device to the processor is one of three types: an address line, a data line, or a control line. The address and data lines are used to select the memory location and to transfer the data, respectively. The control lines tell the memory device whether the processor wants to read or write the location and precisely when the data will be transferred. Unfortunately, one or more of these wires could be improperly routed or damaged in such a way that it is either shorted (for example, connected to another wire on the board) or open (not connected to anything). These problems are often caused by a bit of solder splash or a broken trace, respectively. Both cases are illustrated in Figure 1.

Problems with the electrical connections to the processor will cause the memory device to behave incorrectly. Data may be stored incorrectly, stored at the wrong address, or not stored at all. Each of these symptoms can be explained by wiring problems on the data, address, and control lines, respectively.

If the problem is with a data line, several data bits may appear to be "stuck together" (for example, two or more bits always contain the same value, regardless of the data transmitted). Similarly, a data bit may be either

Listing 2: Address Bus Test

```
datum *
memTestAddressBus(volatile datum * baseAddress, unsigned long nBytes)
{
unsigned long addressMask = (nBytes - 1);
unsigned long offset;
unsigned long testOffset;
datum pattern = (datum) 0xAAAAAAAA;
datum antipattern = (datum) 0x55555555;
/*
* Write the default pattern at each of the power-of-two offsets.
*/
for (offset = sizeof(datum); (offset & addressMask) != 0; offset
<
<
= 1)
{
baseAddress[offset] = pattern;
}
/*
* Check for address bits stuck high.
*/
testOffset = 0;
baseAddress[testOffset] = antipattern;
for (offset = sizeof(datum); (offset & addressMask) != 0; offset
<
<
= 1)
{
if (baseAddress[offset] != pattern)
{
return ((datum *) &baseAddress[offset]);
}
}
baseAddress[testOffset] = pattern;
/*
* Check for address bits stuck low or shorted.
*/
for (testOffset = sizeof(datum); (testOffset & addressMask) != 0;
testOffset
<
<
= 1)
{
baseAddress[testOffset] = antipattern;
for (offset = sizeof(datum); (offset & addressMask) != 0; offset
<
<
= 1)
{
if ((baseAddress[offset] != pattern) && (offset != testOffset))
{
return ((datum *) &baseAddress[testOffset]);
}
}
baseAddress[testOffset] = pattern;
}
return (NULL);
} /* memTestAddressBus() */
```

"stuck high" (always 1) or "stuck low" (always 0). These problems can be detected by writing a sequence of data values designed to test that each data pin can be set to 0 and 1, independently of all the others.

If an address line has a wiring problem, the contents of two memory locations may appear to overlap. In other words, data written to one address will actually overwrite the contents of another address instead. This happens because an address bit that is shorted or open will cause the memory device to see an address different than the one selected by the processor.

Another possibility is that one of the control lines is shorted or open. Although it is theoretically possible to develop specific tests for control line problems, it is not possible to describe a general test for them. The operation of many control signals is specific to either the processor or memory architecture. Fortunately, if there is a problem with a control line, the memory will probably not work at all, and this will be detected by other memory tests. If you suspect a problem with a control line, it is best to seek the advice of the board's designer before constructing a specific test.

Missing memory chips

A missing memory chip is clearly a problem that should be detected. Unfortunately, due to the capacitive nature of unconnected electrical wires, some memory tests will not detect this problem. For example, suppose you decided to use the following test algorithm: write the value 1 to the first location in memory, verify the value by reading it back, write 2 to the second location, verify the value, write 3 to the third location, verify, and so on. Since each read occurs immediately after the corresponding write, it is possible that the data read back represents nothing more than the voltage remaining on the data bus from the previous write. If the data is read back too quickly, it will appear that the data has been correctly stored in memory-even though there is no memory chip at the other end of the bus!

To detect a missing memory chip the test must be altered. Instead of performing the verification read immediately after the corresponding write, it is desirable to perform several consecutive writes followed by the same number of consecutive reads. For example, write the value 1 to the first location, 2 to the second location, and 3 to the third location, then verify the data at the first location, the second location, and so on. If the data values are unique (as they are in the test just described), the missing chip will be detected: the first value read back will correspond to the last value written (3), rather than the first (1).

Improperly inserted chips

If a memory chip is present but improperly inserted in its socket, the system will usually behave as though there is a wiring problem or a missing chip. In other words, some number of the pins on the memory chip will either not be connected to the socket at all or will be connected at the wrong place. These pins will be part of the data bus, address bus, or control wiring. So as long as you test for wiring problems and missing chips, any improperly inserted chips will be detected automatically.

Developing a test strategy Before going on, let's quickly review the types of memory problems we must be able to detect. Memory chips only rarely have internal errors, but, if they do, they are probably catastrophic in nature and will be detected by any test. A more common source of problems is the circuit board, where a wiring problem may occur or a memory chip may be missing or improperly inserted. Other memory problems can occur, but the ones described here are the most common.

By carefully selecting your test data and the order in which the addresses are tested, it is possible to detect all of the memory problems described above. It is usually best to break your memory test into small, single-minded pieces. This helps to improve the efficiency of the overall test and the readability of the code. More specific tests can also provide more detailed information about the source of the problem, if one is detected.

I have found it is best to have three individual memory tests:

Listing 3: Device Test

```
datum *
memTestDevice(volatile datum * baseAddress, un-
signed long nBytes)

{
unsigned long offset;
unsigned long nWords = nBytes / sizeof(datum);
datum pattern;
datum antipattern;
/*
* Fill memory with a known pattern.
*/
for (pattern = 1, offset = 0; offset
<
nWords; pattern++, offset++)
{
baseAddress[offset] = pattern;
}
/*
* Check each location and invert it for
the second pass.
*/
for (pattern = 1, offset = 0; offset
<
nWords; pattern++, offset++)
{
if (baseAddress[offset] != pattern)
{
return ((datum *) &baseAddress[offset]);
}
antipattern = ~pattern;
baseAddress[offset] = antipattern;
}
/*
* Check each location for the inverted pattern
and zero it.
*/
for (pattern = 1, offset = 0; offset
<
nWords; pattern++, offset++)
{
antipattern = ~pattern;
if
(baseAddress[offset] != antipattern)
{
return ((datum *) &baseAddress[offset]);
}
baseAddress[offset] = 0;
}
return (NULL);
} /* memTestDevice() */
```

a data bus test, an address bus test, and a device test. The first two tests detect electrical wiring problems and improperly inserted chips, while the third is intended to detect missing chips and catastrophic failures. As an unintended consequence, the device test will also uncover problems with the control bus wiring, though it will not provide useful information about the source of such a problem.

The order in which you execute these three tests is important. The proper order is: data bus test first, followed by the address bus test, and then the device test. That's because the address bus test assumes a working data bus, and the device test results are meaningless unless both the address and data buses are known to be good. If any of the tests fail, you should work with the board's designer to locate the source of the problem. By looking at the data value or address at which the test failed, he or she should be able to quickly isolate the problem on the circuit board.

Data bus test

The first thing we want to test is the data bus wiring. We need to confirm that any value placed on the data bus by the processor is correctly received by the memory device at the other end. The most obvious way to test that is to write all possible data values and verify that the memory device stores each one successfully. However, that is not the most efficient test available. A faster method is to test the bus one bit at a time. The data bus passes the test if each data bit can be set to 0 and 1, independently of the other data bits.

A good way to test each bit independently is to perform the so-called "walking 1's test." Table 1 shows the data patterns used in an 8-bit version of this test. The name of this test comes from the fact that a single data bit is set to 1 and "walked" through the entire data word. The number of data values to test is the same as the width of the data bus. This reduces the number of test patterns from 2n to n, where n is the width of the data bus.

Since we are testing only the data bus at this point, all of the data values can be written to the same address. Any address within the memory device will do. However, if the data bus splits as it makes its way to more than one memory chip, you will need to perform the data bus test at multiple addresses, one within each chip.

To perform the walking 1's test, simply write the first data value in the table, verify it by reading it back, write the second value, verify, and so on. When you reach the end of the table, the test is complete. It is okay to do the read immediately after the corresponding write this time because we are not yet looking for missing chips. In fact, this test provides meaningful results even if the memory chips are not installed.

The function memTestDataBus(), in Listing 1, shows how to implement the walking 1's test in C. It assumes that the caller will select the test address, and tests the entire set of data values at that address. If the data bus is working properly, the function will return 0. Otherwise it will return the data value for which the test failed. The bit that is set in the returned value corresponds to the first faulty data line, if any.

Address bus test

After confirming that the data bus works properly, you should next test the address bus. Remember that address bus problems lead to overlapping memory locations. Many possible addresses could overlap. However, it is not necessary to check every possible combination. You should instead follow the example of the data bus test above and try to isolate each address bit during testing. You just need to confirm that each of the address pins can be set to 0 and 1 without affecting any of the others.

The smallest set of addresses that will cover all possible combinations is the set of "power-of-two" addresses. These addresses are analogous to the set of data values used in the walking 1's test. The corresponding memory locations are 0001h, 0002h, 0004h, 0008h, 0010h, 0020h, and so on. In addition, address 0000h must also be tested. The possibility of overlapping locations makes the address bus test harder to implement. After writing to one of the addresses, you must check that none of the others has been overwritten.

It is important to note that not all of the address lines can be tested in this way. Part of the address-the leftmost bits-selects the memory chip itself. Another part-the rightmost bits-may not be significant if the data bus width is greater than eight bits. These extra bits will remain constant throughout the test and reduce the number of test addresses. For example, if the processor has 32 address bits, it can address up to 4GB of memory. If you want to test a 128K block of memory, the 15 most-significant address bits will remain constant. 1 In that case, only the 17 rightmost bits of the address bus can actually be tested.

To confirm that no two memory locations overlap, you should first write some initial data value at each power-of-two offset within the device. Then write a new value-an inverted copy of the initial value is a good choice-to the first test offset, and verify that the initial data value is still stored at every other power-of-two offset. If you find a location, other than the one

Table 2: Data values for an increment test

Memory offset	Binary value	Inverted value
000h	0000001	11111110
001h	0000010	11111101
002h	0000011	11111100
003h	0000100	11111011
...	...	...
0FEh	1111111	00000000
0FFh	00000000	11111111

Listing 4: Putting it all together

```
int
memTest(void)
{
#define BASE_ADDRESS (volatile datum *)
0x00000000
#define NUM_BYTES (64 * 1024)
if ((memTestDataBus(BASE_ADDRESS) != 0) ||
 (memTestAddressBus(BASE_ADDRESS, NUM_BYTES) !=
NULL) ||
 (memTestDevice(BASE_ADDRESS, NUM_BYTES) !=
NULL))
{
return (-1);
}
else
{
return (0);
}
} /* memTest() */
```

just written, that contains the new data value, you have found a problem with the current address bit. If no overlapping is found, repeat the procedure for each of the remaining offsets.

The function memTestAddressBus(), in Listing 2, shows how this can be done in practice. The function accepts two parameters. The first parameter is the base address of the memory block to be tested and the second is its size, in bytes. The size is used to determine which address bits should be tested. For best results, the base address should contain a 0 in each of those bits. If the address bus test fails, the address at which the first error was detected will be returned. Otherwise, this function returns NULL to indicate success.

Device test

Once you know that the address and data bus wiring are working, it is necessary to test the integrity of the memory device itself. The thing to test is that every bit in the device is capable of holding both 0 and 1. This is a fairly straightforward test to implement, but takes significantly longer to execute than the previous two.

For a complete device test, you must visit (write and verify) every memory location twice. You are free to choose any data value for the first pass, so long as you invert that value during the second. And since there is a possibility of missing memory chips, it is best to select a set of data that changes with (but is not equivalent to) the address. A simple example is an "increment test."

The data values for the increment test are shown in the first two columns of Table 2. The third column shows the inverted data values used during the second pass of this test. The latter represents a decrement test. There are many other possible choices of data, but the incrementing data pattern is adequate and easy to compute.

The function memTestDevice(), in Listing 3, implements just such a two-pass increment/decrement test. It accepts two parameters from the caller. The first parameter is the starting address and the second is the number of bytes to be tested. These parameters give the user maximum control over which areas of memory will be overwritten. The function will return NULL on success. Otherwise, the first address containing an incorrect data value is returned.

Putting it all together

To make our discussion more concrete, let's consider a practical example. Suppose that we wanted to test a 64K chunk of SRAM at address 00000000h. To do this, we call each of the three test routines in the proper order, as shown in Listing 4. In each case, the first parameter is the base address of the memory block. If the width of the data bus is greater than eight bits, a couple of modifications are required.

If any of the individual memory test routines returns a nonzero (or non-NULL) value, you might turn on a red LED to visually indicate the error. Otherwise, after all three tests have completed successfully, you might turn on a green LED. In the event of an error, the test routine that failed will return some information about the problem encountered. This information can be useful when communicating with the hardware designer or technician about the nature of the problem. However, it is visible only if we are running the test program in a debugger or emulator.

In most cases, you would simply download the entire suite and let it run. Then, if and only if a memory problem is found, would you need to use a debugger to step through the program and examine the individual function return codes and contents of the memory device to see which test failed and why.

Unfortunately, it is not always possible to write memory tests in a high-level language. For example, the C language requires the use of a stack. But a stack itself requires working memory. This might be reasonable in a system with more than one memory device. For example, you might create a stack in an area of RAM that is already known to be working, while testing another memory device. In a situation such as this, a small SRAM could be tested from assembly and the stack could be created there afterward. Then a larger block of DRAM could be tested using a better test suite, like the one shown here. If you cannot assume enough working RAM for the stack and data needs of the test program, then you will need to rewrite these memory test routines entirely in assembly language. Another option is to run the memory test program from an in-circuit emulator. In this case, you could choose to place the stack in an area of the emulator's own internal memory. By moving the emulator's internal memory around in the target memory map, you could systematically test each memory device on the target.

The need for memory testing is most apparent during product development, when the reliability of the hardware and its design are still unproven. However, memory is one of the most critical resources in any embedded system, so it may also be desirable to include a memory test in the final release of your software. In that case, the memory test, and other hardware confidence tests, should be run each time the system is powered-on or reset. Together, this initial test suite forms a set of hardware diagnostics. If one or more of the diagnostics fail, a repair technician can be called in to diagnose the problem and repair or replace the faulty hardware.

References

1. 128K is 1/32,768th of the total 4GB address space.

*This article is adapted from material in Chapter 6 of the book Programming Embedded Systems in C and C++ (ISBN 1-56592-354-5). It is printed with the permission of O'Reilly & Associates, Inc. The source code for these memory tests is placed into the public domain and is available in electronic form at www.embedded.com/code.html .

Email Send inquiry