1. Introduction

There are a number of reasons why computers can give incorrect values, and the results of these errors can range from minor to catastrophic. Some of the major sources of errors can be summarized below:

1. Programming errors
2. Specification errors
3. Hardware errors
4. Communication errors
5. User errors
6. Overflow errors
7. Roundoff errors
8. Specification drift

Mike Quinn gives a number of examples of failures, but doesn’t describe them this way.

In these notes I’ll mainly just give examples of the different types of errors.

2. Programming Errors

Obviously any programming errors can cause applications to fail and give false results. It is assumed that no large program will ever be error free, and so the hope is that the errors in the code will not be too catastrophic. The goal when programming is to attempt to
meet the problem specifications, but all programmers should realize that their code will never do this completely.

The role of testing, which should be designed by the system specifiers (or even better a separate group) and never by the programmers, is to find as many programming errors as possible. Programmers shouldn’t know what will be in the test suite, or inevitably they will just ensure that their programs satisfy the suite.

Typically software, like hardware, exhibits the bathtub effect, where initially a large number of bugs are found, but the number of bugs found per week, or whatever, drops off fairly rapidly and flattens out, hopefully to a fairly small number. After a while, however, bug reports will start to rise again, usually because users, once they get familiar with a product, try to use it for all sorts of additional things which weren’t stressed by early users.

Another annoying thing that happens is that large programs are so inherently complex that fixing one bug can often cause new bugs to appear in unexpected places. The use of structured programming techniques, and in particular programming without uncontrolled gotos, and subsequently of object oriented programming techniques, has reduced this problem somewhat.

3. Specification Errors

Problem specifications describe, in detail, what a program should do, and they are written before anyone starts to program. They define what each major component of the program should do, and also define the interfaces between the components. Programmers must follow specifications and so must assume that the specifications are correct and consistent. If there are errors in the program specifications then automatically there will be errors on the program. This is exacerbated because the testing suites will also be based on the specifications, and so they will fail to find any problems.

4. Hardware Errors

Hardware errors that cause incorrect output are relatively rare (usually they just cause some kind of crash), and so other reasons are usually far more important in terms of giving incorrect results.

There are three major types of hardware errors that can occur, design errors, intermittent errors, and component failures.

Design errors that get past manufacturing and testing are unusual, but when they do occur they tend to lead to bad publicity for the vendor. A classic case was when Intel introduced the Pentium. It and its predecessor, the 486, were unlike most of their predecessors in that they had a math coprocessor which performed floating point (real)
operations in parallel. After it had been in use for a while it was found that the Pentium didn’t always correctly perform division. Most of the time it got it right, and when it did make mistakes they were small, but for some applications the errors were important. The first example publicized was the division 4195835/3145727 where the Pentium came up with the result 1.33374 instead of 1.33382, which is correct. Intel pointed out that an error was likely to occur in only about one in nine billion divides, but by then the only news was that the Pentium couldn’t do its math correctly. IBM later responded that for random pairs of numbers the Intel estimate was correct, but that the numbers that caused the errors appeared at higher frequency than random, and so Intel was underestimating the problem.

Intel announced that they would only replace an individual’s Pentium processor if the individual could justify why they needed to perform very accurate floating point calculations. This decision was heavily criticized, and even made it onto the late night talk show introductions, and finally Intel did what they should have done as soon as the problem was noticed, and replaced the Pentium for anyone who asked.

Intermittent errors, which just pop up occasionally, are the worst to deal with, because they are almost impossible to find, because if they occur rarely enough then when you test for them they might not occur.

Hardware component failures might crash the system, or lose you to lose all of your data, but they don’t usually cause applications to come up with incorrect results.

5. Communication Errors

As data gets transmitted around the world on networks, errors are bound to occur due to limitations on the networking technology. Many of these errors can be detected, and even fixed, using check bits attached to each word that is transmitted. In their simplest form this can just be a parity bit following each group of eight bits transmitted, which could be 1 if the preceding eight bits contained an odd number of 1’s and a 0 if they contained an even number. So if, for example, 10010100 appears over the network, then a transmission error is recognized and a request is made to retransmit the whole packet of information that contains this 9-bit byte. This works well for single bit errors, but if, for example, there are two errors in the byte then no error will be detected by the receiving computer.

With enough extra bits one can detect and even correct multiple bit errors, but transmitting all of this obviously slows down the transmission, which leads to some interesting trade-offs.

There can also be errors based on the underlying design technology. E.g., when my computer sets up a TCP/IP (Internet standard) connection with another computer there is a three-way handshake that occurs, which includes me sending a randomly generated number that will be the ID of my first packet, and the other computer selecting, again
randomly, a packet number for their first response. To play safe we each respond with an acknowledgment of the other’s number to make sure that the numbers were transmitted properly. After the first packet each following packet has an ID number which is one greater that its predecessor. Most of this works well. E.g., if I receive a packet numbered 2045612 and the next one is numbered 2045615, then I know that two packets have been lost in transmission and can request a resend. One potential problem, however, happens when someone else on the network selects a random packet number that overlaps with one of our number sequences, in which case the system falls apart. In individual cases this is very unlikely to occur, but given the incredible number of packets now flying over networks it will occur occasionally.

6. User Errors

Programmers and systems would have a much easier time if they didn’t have to deal with users.

Users can cause systems to give incorrect responses in lots of different ways. The most common are data entry errors and using software outside the boundaries of its design/specifications. If, for example, your name is Jane Smith and someone is entering some negative information about you into the big credit databases, it is very easy for them to pull up the record on the wrong Jane Smith, who will suddenly find that she can’t get any loans and that her credit card interest rates unexpectedly rise. There are constantly reports in the press about people who have been arrested unfairly, receive electricity bills for millions of dollars, find money missing from (or added to) their bank accounts, and so on. Most of these are caused by data entry errors somewhere.

If software is used outside its specifications, then really well designed software will attempt to detect this and print out an appropriate error message. But an accounting program might never guess, for example, that a manager will attempt to terminate an employee by just changing their salary to $0, in which case the computer might continue to give the terminated employee a number of benefits that are standard in the company like a health club membership, or whatever.

When I was at WSU a student took 25 credits and got A’s in all of them. This should have been good but he got a letter saying that he was under suspension warning because of his 0.00 GPA that semester. WSU rules limited students to 20 credits per semester, but the rule wasn’t enforced by the computer. However the record program multiplied the number of credits in each course by the grade (4 for A, 3 for B, etc.) and added the results to get the grade points earned for the semester. Knowing that this shouldn’t ever exceed 80 (20 credits of A) the program efficiently stored this using just two decimal digits. Our poor student got 100 grade points (25 credits at 4 each) and so when this was stored in two digits it just kept the 00 and lost the leading 1, causing the suspension warning letter.
7. Overflow Errors

Computers only maintain a certain number of bits to store numbers. Overflows can occur with either integers or reals (floats), but are most common with integers. These are also the easiest to describe so I’ll just stay with integer overflows.

Integers will usually be stored as 32 bits. Assuming that your computer does this then it will be able to store integers between -2,147,483,648 and +2,147,483,647, which is \(2^{31}\) to \(2^{31} - 1\). When doing this the leftmost bit is 1 for negative numbers and 0 for positive numbers, and the remaining 31 bits are used to store the absolute value of the number.

Now say that we run the C program:

```c
int main()
{
    int num = 2000000000;
    printf("%i + %i = %i
", num, num, num+num);
}
```

Don’t worry about the details of C. What this program will do is output the sum of 2,000,000,000 and 2,000,000,000, and so what you would like to see is an output:

\[2000000000 + 2000000000 = 4000000000\]

but what you actually get is

\[2000000000 + 2000000000 = -294967296\]

which is obviously incorrect. What has happened is an overflow. To see what has happened, look at the comparable situation in the more familiar decimal number system. Say we add 60 to 60. Obviously we need a third digit, the 1 on the left, when we get the result 120. Similarly, if we add two 31 bit numbers (2,000,000,000 in decimal is 1,110,111,001,101,011,001,010,000,000,000 in binary) we get a 32 bit number which begins with 1. However remember that since the leftmost bit is a 1 the computer will recognize this as a negative number, which is how we got the negative result -294,967,296. A problem like this is called an overflow, because we’ve gone past the limit on the size of a positive integer.

8. Roundoff Errors

These errors are tiny errors that can build up because real numbers are usually infinite in representation, but are stored as a finite approximation on the computer.

E.g., consider the expression \(1/3 * 3\). In a good world, this would be 1, but in the computer it will be the binary equivalent of 0.9999999, since 1/3 will be something like 0.3333333. Careful programmers can detect this sort of problem, but a worse situation can occur where roundoff errors slowly build up, which can lead to much larger errors.
9. Specification Drift

As programs are used, users always want to add new features. So what really happens is that the specifications for the program have changed. Usually this occurs as a large number of small changes, and so as much as possible the programmers have to reuse existing code and make small modifications when needed.

When I was at WSU we went from an A, B, C, … grading system to an A, A-, B+, B, B-, C+, … grading system. Obviously this led to some significant changes in the student record and grading systems. A few years later we found that one of our majors had an overall GPA of something like 4.02, and it was impossible to be above 4.00. We looked over his transcript, and he had received straight A’s, so obviously it should have been 4.00, not 4.02. We got fascinated and delved deeper. We finally found that one semester he’d been given a B+ grade, but this had been changed to an A by the instructor after finding a grading error. The parts of the old code that were still being used had changed the letter part from B to A, but hadn’t known that there was still a + around that should have been deleted, so computed his GPA with an A+, even though that grade didn’t exist. The system programmers knew that this must have occurred a number of times but hadn’t been noticed because the change in GPA was so small. They took the expedient approach of fixing the problem in the future, but never chasing down the previous errors.

A much more deadly example of this was the cause of many of the Therac-25 problems, since they inappropriately reused a lot of stable software from previous Therac systems.