» General Programming » Algorithms & Recipes » Algorithms

CRC32: Generating a checksum for a file

Licence
First Posted **18 Dec 2001**
Views **416,893**
Bookmarked **143 times**

By **Brian Friesen** | 18 Dec 2001

VC6 VC7 Win2K WinXP Dev Intermediate

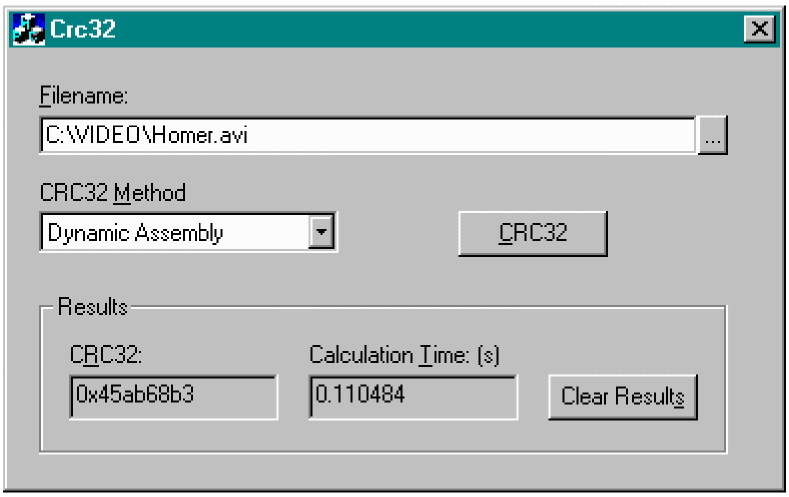
How to generate a CRC32 based on a file

Article Browse Code Stats Revisions Alternatives

See Also
- More like this
- More by this author

- Download source files - 17 Kb
- Download demo project - 8 Kb

Add your own alternative version



Introduction

Recently I wrote a program in which I wanted to generate a **CRC** for a given file. I did some checking on the web for sample **CRC** code, but found very few algorithms to help me. So I decided to learn more about **CRC**s and write my own code. This article describes what a **CRC** is, how to generate them, what they can be used for, and lastly source code showing how it's done.

Redundancy Check (depending on who you ask). A **CRC** is a "digital signature" representing data. The most common **CRC** is **CRC**32, in which the "digital signature" is a 32-bit number. The "data" that is being **CRC**'ed can be any data of any length; from a file, to a string, or even a block of memory. As long as the data can be represented as a series of bytes, it can be **CRC**'ed. There is no single **CRC** algorithm, there can be as many algorithms as there are programmers. The ideal **CRC** algorithm has several characteristics about it. First, if you **CRC** the same data more than once, you must get the same **CRC** every time. Secondly, if you **CRC** two different pieces of data you should get two very different **CRC** values. If you **CRC** the same data twice, you get the same digital signature. But if you **CRC** data that differs (even by a single byte) then you should get two very different digital signatures. With a 32-bit **CRC** there are over 4 billion possible **CRC** values. To be exact that's 2^{32} or 4,294,967,296. With that many **CRC** values it's not difficult for every piece of data being **CRC**'ed to get a unique **CRC** value. However, it is possible for spurious hits to happen. In other words two completely different pieces of data can have the same **CRC**. This is rare, but not so rare that it won't happen.

Why use CRCs

Most of the time **CRC**s are used to compare data as an integrity check. Suppose there are two files that need to be compared to determine if they are identical. The first file is on *Machine A* and the other file is on *Machine B*. Each file is a rather large file (say 500 MB), and there is no network connection between the two machines. How do you compare the two files? The answer is **CRC**. You **CRC** each of the two files, which gives you two 32-bit numbers. You then compare those 32-bit numbers to see if they are identical. If the **CRC** values are different, then you can be 100% guaranteed that the files are not the same. If the **CRC** values are the same, then you can be 99% sure that the files are the same. Remember, because spurious hits can happen you cannot be positive that the two files are identical. The only way to be positive they are the same is to break down and do a comparison one byte at a time. But **CRC**s offer a quick way to be reasonably certain that two files are identical.

How to generate CRCs

Generating **CRC**s is a lot like cryptography in that involves a lot of mathematical theories. Since I don't fully understand it myself, I won't go into a lot of those details here. Instead I'll focus on how to program a **CRC** algorithm. Once you know how the algorithm works you should be able to write a **CRC** algorithm in any language on any platform. The first part of generating **CRC**s is the **CRC** lookup table. In **CRC**32 this is a table of 256 specific **CRC** numbers. These numbers are generated by a polynomial (the computation of these numbers and what polynomial to use are part of that math stuff I'm avoiding). The next part is a **CRC** lookup function. This function takes two things, a single byte of data to be **CRC**'ed and the current **CRC** value. It does a lookup in the **CRC** table according to the byte provided, and then does some math to apply that lookup value to the given **CRC** value resulting in a new **CRC** value. The last piece needed is the actual data that is to be **CRC**'ed. The **CRC** algorithm reads the first byte of data and calls the **CRC** lookup function which returns the **CRC** value for that single byte. It then calls the **CRC** lookup function with the next byte of data and passes the previous **CRC** value. After the second call, the **CRC** value represents the **CRC** of the first two bytes. You continuously call the **CRC** lookup function until all the bytes of the data have been processed. The resulting value is the **CRC** for the

Related Articles

A File Checksum Shell Menu Extension DLL
An Analysis of the Windows PE Checksum Algorithm
Cyclic Redundancy Check (CRC32) HashAlgorithm
CRC Encoding
A Checksum Algorithm
ZipBuilder - A zip package creation utility for collections of large files
Very secure method to save and restore registry
Verifying .MD5 file verification databases
SimpleZip
Managed C++ wrapper for ZLib
Error Detection Based on Check Digit Schemes
XReverse - Reverse a text file using memory-mapped files
CRC_32
CFil I f A A I f h i

In this sample program I wanted to show that there are many different ways of generating **CRC**s. There are over 8 different **CRC** functions, all based on the above steps for generating **CRC**s. Each function differs slightly in it's intended use or optimization. There are four main **CRC** functions, each described below. There are also two separate **CRC** classes, but more on that later. And lastly there are a few helper functions that **CRC** strings.

C++ Streams: The first function represents the simplest **CRC** function. The file is opened using the C++ stream classes (ifstream). This function uses nothing but standard C++ calls, so this function should compile and run using any C++ compiler on any OS.

Win32 I/O: This function is more optimized in that it uses the Win32 API for file I/O; CreateFile, and ReadFile. This will speed up the processing, but by using the Win32 API the code is no longer platform independent.

Filemaps: This function uses memory mapped files to process the file. Filemaps can be used to greatly increase the speed with which files are accessed. They allow the contents of a file to be accessed as if it were in memory. No longer does the programmer need to call ReadFile and WriteFile.

Assembly: The final **CRC** function is one that is optimized using Intel Assembly. By hand writing the assembly code the algorithm can be optimized for speed, although at the sacrifice of being easy to read and understand.

Those are the four main **CRC** functions. But there are actually two versions of each function. There are two classes, CCrc32Dynamic and CCrc32Static, each of which have the above four functions for a total of eight. The only difference between the static and dynamic classes is the **CRC** table. With the static class the **CRC** table and all the functions in the class are static. The trade off is simple. The static class is simpler to use, but the dynamic class uses memory more efficiently because the **CRC** table (1K in size) is only allocated when needed.

⊟ Collapse | Copy Code

```
// Using the static class is as easy as one line of code
dwErrorCode = CCrc32Static::FileCrc32Assembly(m_strFilename,
dwCrc32);

// Whereas there is more involved when using the dynamic class
CCrc32Dynamic *pobCrc32Dynamic = new CCrc32Dynamic;
pobCrc32Dynamic->Init();
dwErrorCode = pobCrc32Dynamic->FileCrc32Assembly(m_strFilename,
dwCrc32);
pobCrc32Dynamic->Free();
delete pobCrc32Dynamic;
```

Whenever you calculate a **CRC** you need to take into account the speed of the algorithm. Generating **CRC**s for files is both a CPU and a disk intensive task. Here is a table showing the time it took to **CRC** three different files. The columns are the different file sizes, the rows are the different **CRC** functions, and the table entries are in seconds. The system these numbers were captured on is a dual Pentium III at 1 GHz with a 10,000 RPM SCSI Ultra160 hard drive.

	44 Kb	34 Mb	5 Gb
C++ Streams	0.0013	0.80	125
Win32 I/O	0.0009	0.60	85
Fil	0 0010	0 60	87

Writing Your Own GPS Applications: Part I

32-bit UDP Checksum

Adler-32 Checksum Calculation

CHash 1.5 - An MFC hashing class

CrcStream stream checksum calculator

The Daily Insider

As expected the C++ streams is the slowest function followed by the Win32 I/O. However, I was very surprised to see the filemaps were not faster than the Win32 I/O, in fact they are slower. After I thought about it some, I realized memory mapped files are designed to provide fast random access to files. But when you **CRC** you access the file sequentially. Thus filemaps are not faster, and the extra overhead of creating the "views" of the file are why it's slower. Filemaps do have one advantage that none of the other functions have. Memory mapped files are guaranteed to be able to access files up to the maximum file size in NT which is 2^{64} or 18 exabytes. Although the Win32 I/O may handle files of this size, none of the documentation confirms this. [Note: The largest file I have **CRC**'ed is 40 GB, which all eight functions successfully **CRC**'ed, but took over 10 minutes each.]

If anyone who reads this article knows a way to improve the speed even more, please post the code or email me. Especially if you know of a speed improvement for the assembly code. I will bet there are further optimizations that can be made to the assembly code. After all I don't know Intel Assembly very well, therefore I'm sure there are optimizations I don't know about.

License

About the Author

Brian Friesen

Web Developer

United States

Member

Article Top   Rate this: *Poor* ○ ○ ○ ○ ○ *Excellent* Vote

Comments and Discussions

New Message

Search this forum Go

Profile popups Noise Medium Layout Normal Per

Thank you very much!	**yulin11**	**17:59 2 Oct '08**
How many errors..	**Rikard Astrof**	**18:01 25 Feb '08**
Re: How many errors..	supercat9	3:40 15 Apr '08
Licensing	**allen_ellison**	**11:33 16 Jan '08**
Computing File CRC in VBScript	**Milind Mehendale**	**14:06 26 Nov '07**
How to compile?	**maxsubzero**	**8:51 31 Oct '06**
Re: How to compile?	VEMS	4:36 10 Jan '07
Re: How to compile?	maxsubzero	6:05 21 Jan '07
Re: How to compile?	chris_liush	14:33 23 Oct '07
Re: How to compile?	ispeedonthe405	8:50 17 Nov '07
Re: How to compile?	tianjianii	18:29 27 Jun '08
Re: How to compile?	Tim Stubbs	22:33 6 Aug '09
link error	**JakeFront**	**1:31 20 Sep '06**
Re: link error	Brian Friesen	7:44 20 Sep '06
Re: link error	JakeFront	16:58 20 Sep '06
Re: link error	aldasp	23:15 28 Nov '06
Re: link error	JakeFront	0:02 29 Nov '06
Should I Change CRC Table?	**OffLineR**	**16:26 1 Feb '06**
Re: Should I Change CRC Table?	aldasp	15:58 29 Nov '06
Thanks	**OffLineR**	**16:20 1 Feb '06**
Different CRC	**Sniper167**	**5:56 23 Jan '06**
Re: Different CRC	Brian Friesen	7:09 23 Jan '06
Re: Different CRC	VEMS	4:49 10 Jan '07
Re: Different CRC	awneil	22:30 14 Jul '08
this is not the way to use CRC	**jt_roxas**	**9:30 11 Dec '05**

Last Visit: 13:03 26 Feb '12 Last Update: 9:56 3 Mar '12 **1** 2 3 4 Next »

General News Suggestion Question Bug Answer Joke Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.