Checksum Calculation – CRC32

By Thomas Maul, 4D Germany

Technical Note 01-17
Technical Notes for 01-04 April 2001

Abstract

The checksum is a very useful tool to verify the integrity of a file. There are several ways in which the contents of a file can be modified and become corrupt. This includes disk or RAM failure (hardware problem), CD-ROM reading error (medium problem), transfer error (modem), or human error.

Although Ruffin Scott, in TN 99-11 already describes the concept of a CRC checksum, the concepts may be difficult to understand and implement. The purpose of this tech note is to simplify CRC checksum and allow you to implement CRC without the need to understand how CRC works. The tech note includes all the code you need to implement CRC32 in your databases.

Checksums in 4D

Internally, every record in 4D is checked for errors using CRC checksum. If 4D detects a checksum error when reading a record, it displays a dialog “Record is bad. Do you want to delete it or mark it for checking with 4D Tools?” By using checksum 4D guaranties that a record is valid.

Use of checksums in 4D applications

Some data are automatically checked using checksum, while others are not. Here are some examples where checksum is not automatically used.

Blobs

Blobs were introduced into the life of 4D developers with the release of 4D V6. While Blobs are very useful, 4D cannot control the contents of a Blob. This means that it is vital that the developer develops ways or techniques to verify the contents of a Blob. Most developers blindly trust that a Blob still contains the correct data when reading a Blob and react helplessly if reading the Blob fails.

Sending data to customers

Often data such as zip codes, phone numbers, article lists, and so on are part of an application. This data is usually sent as a Blob or as a text import document. Again, most people take for granted that the data received is correct.

Distributed Systems synchronization

Distributed Systems are two or more databases that are physically in different locations. As a result, Distributed Systems need to be synchronized regularly. If this is done via 4D Open, records are automatically checked by 4D. On the other hand, if Blobs are used to exchange data or data is sent as documents via FTP, e-mail, or any other similar means of sending data, it is mandatory to check the validity of the data.

What is a checksum?

You can “invent” a simple checksum system by counting bytes or by adding all bytes together. This system, however, is prone to errors and cannot detect byte exchange or similar errors. There is a free algorithm named Cyclic Redundancy Check or CRC that is a non-linear system. CRC32 is a special sub form optimized for current 32-bit CPU´s. The calculation of the checksum is based on the following polynomial:

$$G(x) = x32 + x26 + x23 + x22 + x16 + x12 + x11 + x10 + x8 + x7 + x5 + x4 + x2 + x + 1$$

Although current machines are relatively fast, this calculation, because of its complex calculation is too slow for large files. An example of a large file would be data that is exported. There is a simpler calculation based on an array, which in interpreted mode is fast enough for large files. The best thing is, this algorithm is still free– waiting for you to implement it into your system!

Code to calculate a checksum in 4D

Here is an example of how to calculate a checksum:

```
C_BLOB(Test_Blob)
$ref:=Open document("")
If (OK=1)
   CLOSE DOCUMENT($ref)
   DOCUMENT TO BLOB(document;Test_Blob)
   CRC_CreateTable
   $checksum:=CRC_Calculate (->Test_Blob)
   ALERT(String($checksum))
End if
```

This code will display an open-file dialog so the user can select a file. The file is read into a Blob and the Blob is passed to the function *CRC_Calculate*, which then returns the checksum. The routine *CRC_CreateTable* calculates an array needed to create the checksum. Because this table always remains the same, if you wanted to create more than one checksum, you would only need to call this method once. In interpreted mode, this method on a 900 MHZ AMD needs only 3 ticks (1/20 of a second) and in complied mode it is too fast to time!

Calculating the checksum, on the other hand, is a little slower. In interpreted mode, on a 900 MHZ AMD, requires 130 ticks or roughly 2 seconds for a 100kb File. For a 3 MB file, it's 4,155 ticks or nearly 70 seconds. In compiled mode, the same 100kb File requires 5 ticks — or less than 1/10 of a second. The 3 MB file, which required 4155 ticks in interpreted mode, requires only 145 ticks in complied mode.

The benefit of this system is tremendous. The cost is two methods and, for a 3 MB export file, 2 seconds, while saving you hours of work trying to figure out why an update works on your machine but does not work after sending it to your customer.

Method CRC_Calculate

The following code is the complete code to calculate the checksum of any document. It needs a longint array with 256 elements with some "helping values". Don't worry; you do not need to understand the contents to use it!

```
` Method CRC_Calculate
C_POINTER($1)
C_LONGINT($0;$value)

$value:=0xFFFFFFFF
$n:=BLOB size($1->)
For ($i;0;$n-1)
   $value:=($value >> 8) ^| CRC_Table{($1->{$i}) ^| ($value & 0x00FF)}
End for
$0:=$value
```

If you have never worked in hex or thought about direct bit manipulation, skip it. However, if you have worked with C or an Assembly language before using 4D, you may be surprised that 4D can

directly display or enter hex values. 4D even contains operators like AND, OR, XOR and bit shift. The 4D programming language manual contains a chapter dedicated to Operators. Under the chapter Operators, bitwise operators explain bit shift. This is quite useful for implementing algorithms with bit shift.

Method CRC_CreateTable

Again, one does not need to understand what the method does to implement it. Simply call the method once before using *CRC_Calculate*. By calling the method ARRAY LONGINT(CRC_Table;255), it creates the necessary longint array CRC_Table with 256 elements, including the 0th element, and filling it with strange values.

Just to give you an idea, here the first few values:

```
0x00000000 0x77073096 0xEE0E612C 0x990951BA
0x076DC419 0x706AF48F 0xE963A535 0x9E6495A3
```

There is no systematic way in which these numbers were generated.

```
ARRAY LONGINT(CRC_Table;255)
C_LONGINT($c)

For ($n;0;255)
   $c:=$n
   For ($k;0;7)
      If (($c & 1)>0)
         $c:=0xEDB88320 ^| ($c >> 1)
      Else
         $c:=$c >> 1
      End if
      CRC_Table{$n}:=$c
   End for
End for
```

To use checksum in your database, simply copy the two methods into your code. Both methods start with CRC_ using one global variable, the array CRC_table.

Where to save the checksum

There are two ways in which a checksum can be saved for later comparison. One is obvious: Simply store the checksum in another variable or object and use the new variable or object for comparison later.

The second is fascinating. After calculating the checksum, put the longint as 4 Byte directly after your document. This is very easy to do with Blobs by using the LONGINT TO BLOB command. If you now calculate the checksum again and use the full document, including the 4 added bytes, the returned checksum is always Zero. This means that at the receiver side, all you do to verify the contents of a file is to create the checksum again. If it's zero, everything is fine. If the checksum is not equal to zero, the document is either damaged or does not end with a CRC32 checksum. Because the last four bytes of the file are part of the checksum, in order to use the file, the last 4 bytes of the document must be thrown out.

Summary

This tech note provides a simple and practical way of calculating a checksum in 4D databases. The technique involves only two simple methods. With a checksum, you can verify the contents of blobs as well as imported and exported documents of all types.