The HashTools Component

By David Adams

Technical Note 05-43

Overview

Hashing is a technique for rendering a block of data, such as a string, BLOB, picture, or document, as a single longint. Hashes can be used as check sums or to optimize lookup and query operations. The HashTools component implements nine different hashing algorithms and provides code for hashing text, pictures, BLOBs, and documents. In addition, the component includes code to perform hash-optimized searches on text, BLOB, and picture fields. This technical note describes the public and protected methods in the component. Several complementary technical notes describe other applications for hashing in 4th Dimension or discuss details of the design and implementation of the HashTools code:

- Technical Note 05-44 **Optimizing Searches with Hashes** describes the source code of the HashTools component and how hashing can optimize text and BLOB queries. Detailed test results are provided to compare the performance characteristics of different hashing algorithms. The sample database from that note is included here to illustrate various HashTools features.
- Technical Note 05-41 **Case-Sensitive Operations in 4th Dimension** discusses how to perform case-sensitive searches in 4th Dimensions using several different techniques, including using the HashTools component.
- Technical Note 05-42 **Scanning Text and BLOBs Efficiently** explores in detail how to design 4th Dimension code to analyze text and BLOB values most efficiently for best speed and best memory conservation.

Why a Component?

Components provide a way of packaging a collection of 4th Dimension resources for easy distribution. Some developers avoid building or using components because of problems in historical versions of 4th Dimension or because the code of protected methods is not visible once they are installed in a database. In counterpoint to these potential drawbacks, packaging the HashTools code as a component delivers several benefits:

- **Reduced Complexity**: The source code includes over 50 methods, most of which should not be called directly. The component exposes about 10 methods, half of which are devoted to error management, compilation, and documentation.

- **Simplified Updating**: The HashTools routines are used in several related technical notes and databases, as mentioned. Using a component makes it easy to keep the hashing code in these systems up-to-date. The same is true of any databases in which you install HashTools.
- **Efficient Error Testing**: Internally, the hashing functions need to test that parameters, such as pointers and document references, are valid. Requiring each low-level method, such as *HashText_RS*, to perform these checks renders the routines larger and more complex. Instead, the low-level routines are made private and do no parameter testing of any kind. Protected methods, such as *HashTools_HashText*, act as gateways to the private methods. The gateway methods test parameters comprehensively before calling the private methods. This strategy lets the component protect thoroughly against bad inputs without redundancy and with the least complexity possible.

Note *Technical Note 05-44* **Optimizing Searches with Hashes** *includes the original source code for the HashTools component, if you want or need to rewrite or extend the existing component.*

Compilation

The code in the HashTools component is designed to be run compiled, not interpreted. The component includes complete declarations for all variables, arrays, and parameters used. You may compile with any of 4th Dimensions compilation options, including "all variables are typed".

Internal Documentation

Each visible method in the component includes Explorer comments documenting the method's parameters and a relevant code sample. Additionally, the *HashTools Read Me Public* method describes and documents each method.

Supported Hash Algorithms

There are hundreds of existing text hashing functions. The ideal hashing function for any particular situation depends on the data itself. The HashTools component adapts C code for eight high-quality functions into 4th Dimension code. Additionally, a ninth function called SumBytes is included for didactic purposes. The original C source code, and versions in C++, Object Pascal, and Java, can be found at this Arash Partow's site:

http://www.partow.net/programming/hashfunctions/

The table below, adapted from Arash Patrow's site, briefly describes the nine algorithms supported by the HashTools component.

Algorithm	Notes
AP	A new hashing algorithm designed by Arash Partow, the programmer who provides the sample C code used as the basis for HashTools.
BKDR	An algorithm adapted from ***The C Programming Language*** by Kernighan and Ritchie.
DJB	A highly efficient hashing function developed by Daniel J. Bernstein and first published on the comp.lang.c newsgroup.
ELF	A 32-bit version of the PJW hash.
JS	A bitwise hashing function developed by Justin Sobel.
PJW	An algorithm first developed by Peter J. Weinberger at AT&T Bell Labs.
RS	An optimized version of a hashing function found in Robert Sedgwick's ***Algorithms in C***.
SDBM	A good low-collision algorithm used in many database projects.
SumBytes	A low-quality but simple and easy to understand hash that sums the value of each byte in a string, text, BLOB, picture, or document.

If you need to retrieve the list of valid hashing algorithm names within code, use the *HashTools_GetHashTypeNames* function:

ARRAY TEXT($hashMethodNames;0)
HashTools_GetHashTypeNames(->$hashMethodNames)

Note *You may pass a pointer to a string or a text array to* HashTools_GetHashTypeNames. *If you pass a string array, be sure the elements are at least 8 characters long.*

Selecting a Hashing Algorithm

Hashing algorithms differ in the time they need to create a hash and the number of collisions (duplicate hashes from distinct inputs) they produce. Technical Note 05-44 **Optimizing Searches with Hashes** explains in detail how hashing works, what collisions are, and how to test different algorithms. That note also provides summaries of test results for the nine hashing algorithms when applied to different data sets. Based on those tests, you may consider the AP, BKDR, RS, and SDBM algorithms functionally equivalent for your 4th Dimension projects.

Hashing Values

The HashTools component includes four routines for hashing different types of values. Each routine requires a reference to a value of the correct type, and the name of a valid hashing algorithm, and returns a hash in a longint. The valid hashing algorithms names can be read in code using the *HashTools_GetHashTypeNames* routine. If there is an error, such as a bad pointer being passed in or an unrecognized hash method name, the HashTools component sets an error. See the error management documentation below for details.

HashTools_HashBlob

HashTools_HashBlob (Pointer;Text): Longint
HashTools_HashBlob (->BLOB;"Hash Method"): Hash

Hashes a BLOB and returns a result, as in the example below:

```
C_LONGINT($hash)
$hash:=HashTools_HashBlob(->[Sample]BLOB;"BKDR")
```

Note *The BLOB is passed in by pointer but is duplicated internally to improve performance. If this behavior causes memory problems in your environment, you can rewrite the underlying private hashing routines to operate on pointers.*

HashTools_HashDocument

HashTools_HashDocument (Time;Text): Longint
HashTools_HashDocument (DocRef;"Hash Method"): Hash

Hashes a document and returns a result, as in the example below:

```
C_TIME($docref)
$docref:=Open document("")

If (OK=1)
  CREATE RECORD([StoredDocumentData])
   [StoredDocumentData]Hash:=HashTools_HashDocument ($docref;"AP")
   [StoredDocumentData]Path:=Document
  SAVE RECORD([StoredDocumentData])
  UNLOAD RECORD([StoredDocumentData])

  CLOSE DOCUMENT($docref)
End if
```

The document reference must be to a document that has already been opened or else an error is set. After running *HashTools_HashDocument*, the document is left open and the document position is restored.

Note *The document is hashed by reading in a stream by 32,000 character chunks rather than by copying the entire document into a BLOB. This procedure is relatively quick and enables the* HashTools_HashDocument *routine to hash enormous documents without consuming much memory.*

HashTools_HashPicture

HashTools_HashPicture (Pointer;Text): Longint
HashTools_HashPicture (->Picture;"Hash Method"): Hash

This routine hashes a picture and returns the result, as in the example below:

```
C_LONGINT($hash)
$hash:=HashTools_HashPicture(->[Sample]Picture;"SDBM")
```

Note *The picture is passed in by pointer and is then converted internally to a BLOB and then passed as a parameter, thus consuming extra memory. If this causes memory problems in your environment, you can rewrite the underlying private hashing routines to operate on pointers or on a dedicated process variable.*

HashTools_HashText

HashTools_HashText (Pointer;Text): Longint
HashTools_HashText (->Text;"Hash Method"): Hash

Hashes a string or block of text and returns a result, as in the example below:

```
C_LONGINT($hash)
$hash:=HashTools_HashText (->[Sample]Text;"SDBM")
```

Note *The text is passed in by pointer but is copied locally to avoid scanning an alpha through a pointer. In some versions of 4th Dimension, scanning blocks of text with about 15,000 characters or more can be 100 times slower than expected. See Technical Note 05-42,* **Scanning Text and BLOBs Efficiently***, for more details.*

Finding Records by Stored Hashes

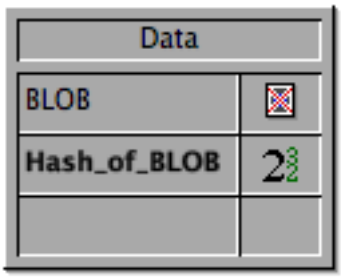
The *HashTools_FindByHash* routine can automatically match alpha, text, BLOB, and picture fields using a stored hash. The parameters and behavior of this method are documented below

HashTools_FindByHash (Pointer;Longint;Pointer;Pointer;Boolean) : Longint
HashTools_FindByHash (->Value to match;Hash to match;->Value field;
->Hash field;{Search selection?}): Hash matches

Special notes on each parameter are included below:

Parameter	#	Type	Notes
Hash matches	$0	Longint	How many records matched the hash? This value is useful during hash analysis but typically not needed during routine use. To find the number of records located, use **Records in selection**.
Value to match	$1	Pointer	Pointer to the alpha, text, BLOB, or picture to match.
Hash to match	$2	Longint	Hash of value to match. You must prepare and supply the hash as *HashTools_FindByHash* doesn't do any hashing.
Value field	$3	Pointer	A pointer to the field that stores the alpha, text, BLOB, or picture you want to find. The type of the value field ($3) and the value to match ($1) must agree.
Hash Field	$4	Pointer	A pointer to the field that stores hashes.
Search selection?	$5	Boolean	Search the current selection? The default value for this optional parameter is **False**, meaning "search the table".

This routine depends entirely on stored and supplied hashes. This routine doesn't do any hashing at all. For this system to work, you must pre-calculate and store the hashes of the field you are interested on searching. As an example, imagine a table that stores large BLOBs, such as the one pictured below:



In this imaginary table structure, the [Data]Hash_of_BLOB field stores the SDBM hash of the contents of the [Data]BLOB field. The sample code below shows how to find values in [Data]BLOB that match a BLOB stored in a local variable named $BlobToMatch_blob:

C_LONGINT($hash)
` Hash the local BLOB using the same method used to hash the stored BLOBs.
$hash:=*HashTools_HashBLOB* (->$BlobToMatch_blob;"SDBM")
HashTools_FindByHash (->BlobToMatch_blob ;$hash;->[Data]BLOB;
->[Data]Hash_of_BLOB)

Internally, *HashTools_FindByHash* routine first performs an indexed search on the [Data]Hash_of_BLOB field, eliminating from consideration all BLOBs that don't have a hash value identical to the hash of the test BLOB. If you are storing unique BLOBs, the indexed search is likely to reduce the selection to only the matching BLOB, if any. Once the indexed search is completed, the *HashTools_FindByHash* routine tests the remaining records to see if the test value and the value stored in the record are

identical. *HashTools_FindByHash* is able to do this with alpha, text, BLOB, and picture fields.

Tip *Triggers are a good place to put the code needed to maintain stored hashes.*

Error Management

The routines in the HashTools component install a custom error handler before performing any work. If you have another error handler installed already, it is restored at the end of any HashTools routine. If an error is encountered by HashTools, the method that encountered the error and an error code are set. You can read these values with the functions documented below.

HashTools_GetLastErrorLocation

HashTools_GetLastErrorLocation (): Text
HashTools_GetLastErrorLocation (): Name of last method to set an error, if any.

HashTools_GetLastErrorCode

HashTools_GetLastErrorCode (): Longint
HashTools_GetLastErrorCode (): Error code or 0, if there was no error.

HashTools_GetErrorText

HashTools_GetErrorText (Longint): Text
HashTools_GetErrorText (Error code): Error text

A string translation of any HashTools error code can be read using the *HashTools_GetErrorText* function. If you want the text of the last error code set, call the code shown below:

HashTools_GetErrorText (*HashTools_GetLastErrorCode*)

Defined Error Strings

The table below lists all defined HashTools errors.

#	Error Text
1	Required parameter(s) not passed to HashTools.
2	Bad hash method type passed to HashTools.
3	Bad document reference passed to HashTools.
4	Unrecognized hash method type, HashTools internal error.
5	Nil pointer passed to HashTools.
6	Pointer to wrong data type passed to HashTools, BLOB pointer expected.
7	Pointer to wrong data type passed to HashTools, alpha/text pointer expected.
8	Bad pointer passed to HashTools, pointer to search field expected.
9	Bad pointer passed to HashTools, pointer to hash field expected.
10	Pointer to wrong search field type, string, text, BLOB, or picture field expected.
11	Pointer to wrong hash field type, numeric field expected.

12	Search value and search field types do not agree.
13	Calling HashTools_GetErrorText without the required error code parameter.
14	Bad pointer passed to HashTools, pointer to picture expected.

Summary

The HashTools component provides an easy-to-use suite of hashing functions that can be applied to alphas, text, pictures, BLOBs, and documents. Moreover, the component includes a routine for performing highly-optimized hash-based searches on alpha, text, BLOB, and picture fields. Additional technical notes describe and discuss hashing, applications for hashing, and the design of text or BLOB code, in more detail, including Technical Note 05-44 **Optimizing Searches with Hashes**, Technical Note 05-41 **Case-Sensitive Operations in 4th Dimension**, and Technical Note 05-42 **Scanning Text and BLOBs Efficiently**.