

Keeping the Memory Clean - Part 2

By Walt Nelson, Director of Partner Development

4th Dimension Technical Note 97-33

Technical Notes for 97-09-September 1997

Introduction

At any given time, the performance and stability of any 4D application is affected by the amount of RAM that is available for loading objects into memory, and by the sizes of the blocks of available RAM.

In this two-part technical note, you will learn how 4D uses memory, and what you can do in order to keep the memory clean. In Part 1, we discussed how 4D uses memory, and we gave you some tips for optimizing the memory in your forms. In Part 2, we explain how to write 4D code that helps to keep the memory clean.

What is Clean Memory?

We defined clean memory in Part 1, but the definition is so important that we will repeat it here:

When an object is loaded into memory, 4D needs to have one contiguous block of memory into which to load the object. For example, if the object is a several-page Form that is 350k in size, then 4D needs a single block of RAM that is at least 350k in size, in order to load that form. This is an important point: even if 4D has more than 350k of memory available, 4D cannot load a large object if the memory is fragmented. In such a case, 4D will have to "purge and merge": remove unused objects from memory, and then move the remaining objects around to create a block large enough to load the new object. At best, there will be a delay; at worst, 4D will not be able to create a block large enough to load the object. Obviously, then, the ideal situation is to keep the size of your objects small, and to keep the size of the available memory blocks large. When you have this combination, you have "clean memory."

Writing Memory-Efficient Code

When you write 4D code, you should always keep in mind the memory consequences of your code. Here are some tips for memory-efficient code—code that keeps the memory clean:

Limit the size of your 4D methods

The larger the size of a method in kilobytes, the more difficult it might be for 4D to find a place in memory for the method. In general, you should limit your methods to one or two printed pages. When a method starts to exceed that limit, look for ways to break it down into two or more smaller methods. If you do this, your methods will never be more than a few kilobytes in size, and 4D will seldom have trouble finding a place in RAM to load them.

There is another good reason to keep the size of your methods small: when a method is executing, it is locked in memory—it cannot be purged, it cannot be moved. Generally speaking, the larger your methods, the longer they will take to execute and the longer they will remain locked in memory. When you have several large methods locked in memory, they can cause the memory to become fragmented.

Clear the current selection from memory when you no longer need it

For each global process, for each table, you always have one current selection. The current selection can be empty, or it can contain all the records in the table, or any number of records in between. In terms of memory usage, the size of the current selection will always be: (4 bytes * number of records in selection) + 4 bytes. For example, if the current selection is 1,000

records, it will use 4,004 bytes of memory (approximately 4k); 10,000 records, 40k; 100,000 records, 400k; and so on. As soon as you have finished with a current selection, clear the memory by calling the following command:

```
REDUCE SELECTION([TableName];0)
```

Note: The ALL RECORDS command is optimized. Instead of creating an address table that is four bytes times the number of records in the selection, 4D creates an exception table of the deleted records: all records that are not in the deleted selection will be in the current selection.

Unload the current record from memory when you no longer need it

For each current selection, you may have one current record. (We say "may have" because there are some situations in which you do not have a current record.) If you have a current record, the size of this record can be anywhere from a few bytes to several megabytes, depending on what you are storing in the record. As soon as you have finished with a current record, clear the memory by calling the following command:

```
UNLOAD RECORD([TableName])
```

When you unload a record in this manner, the record is still the current record but it is no longer loaded in memory. Also, the record is no longer locked by the current process; the record is available to be loaded and modified by other processes or other users.

Note: When you show a selection of records in an output form with the MODIFY SELECTION or DISPLAY SELECTION command, 4D is optimized. It does not load a current record unless the user actually double-clicks a record.

Create named selections with the CUT NAMED SELECTION command whenever possible

When you want to create a named selection, you have two options. You can use the COPY NAMED SELECTION command or the CUT NAMED SELECTION command. When you use COPY NAMED SELECTION, you are duplicating the current selection in memory: (4 bytes * records in selection). A 10,000-record named selection will use 40k of memory. If 4D does not have enough memory to duplicate the selection, it returns Error Number -108.

When you use CUT NAMED SELECTION, you are merely creating a 4-byte pointer to the selection. The current selection becomes empty, and the selection becomes a named selection. This operation will always be possible, because it requires only 4 bytes of memory. Therefore, whenever you don't need to maintain both the current selection and the named selection, create named selections with CUT NAMED SELECTION rather than with COPY NAMED SELECTION.

Clear a named selection from memory when you no longer need it

If a named selection was created with the CUT NAMED SELECTION command, it is self-cleaning as soon as you call the USE NAMED SELECTION command—the named selection no longer exists, and the selection becomes the current selection. However, if you created a named selection with the COPY NAMED SELECTION command, it does not clear automatically when you use it. Therefore, you might write the following two lines of code when you use the named selection:

```
USE NAMED SELECTION(SelectionName)
CLEAR NAMED SELECTION(SelectionName)
```

Clear sets as soon as you no longer need them

A set is a presence table consisting of an 8-byte header, plus one bit for each record in the set's table. This presence table includes all records that exist, or have existed, in the table. In other words, deleted records that have not been overwritten will be represented in the table. Every record in the table is either present in the set (bit is "on") or not present in the set (bit is "off"). Even if there is only one record in a set, the set still takes up the same amount of space in memory: 8 bytes + (records in table * 1 bit).

To calculate the size of a set in bytes, use the following formula:

(Records in table divided by 32) + 8 bytes = size of the set in bytes

For example, if there are 100,000 records in the table: $(100,000/32) + 8$ bytes = 3,133 bytes.

To clear a set, call the command:

CLEAR SET (ÒSetNameÓ)

Use local variables whenever possible

A local variable exists only within the method in which it is used. The name of a local variable begins with a dollar sign. During compilation, a local variable table is created for each method. This local variable table is loaded onto the process stack when the method is executed. When the method is completed, the local variables no longer exist. Access to local variables is very fast and very efficient, because the variables are automatically cleared when the method stops executing.

Here are some additional notes about local variables:

¥ An array is a variable; use local arrays whenever possible.

¥ Do not confuse local variables with passed parameters (\$1, \$2, etc.).

¥ Since local variables are not defined for the entire database (only for one specific method), local variables cannot be accessed through pointers.

¥ Since local variables are not defined for the entire database, they cannot be displayed on forms.

Limit the total size of the process variable table

When you create a variable whose name does not begin with a dollar sign (\$) or with the interprocess symbol (<>), that variable is a process variable. When you compile, that variable is added to the process variable table. In a compiled application, the entire process variable table is loaded into memory upon startup. This copy of the table is used by the main process. Each time a new process is created, 4D makes another copy of the process variable table for the exclusive use of the new process. In other words, you will have as many copies of the process variable table as you have processes running. This can consume quite a bit of memory. As a general rule, try to limit the size of the process variable table to 20k or less.

To find the size of the process variable table, turn on the Symbol Table option in 4DÉCompiler.

Open the symbol table with a word processor. At the end of the list of process variables, you will find the size of the process variable table. At the end of the list of interprocess variables, you will find the size of the interprocess variable table.

Here are examples of that information in the 4DÉCompiler symbol table:

```

=====
<>4DCALCPID      Long integer          (M) * On Startup
...
<>WRPLTTWPID     Long integer          (M) * On Startup
Interprocess variables size : 112
=====

=====
ACOURSEBLDG      Text                  1 dimension      (M) 4D_DrawNew
...
ZZ9              Long integer          (F) [Registrations].Input
Process variables size : 1542
=====

```

Here are some ways in which you can limit the size of the process variable table:

- ¥ Use local variables.
- ¥ Use variable naming conventions. For example, name all your Accept buttons ÒbOK.Ó
- ¥ If it is not necessary to have a separate instance of a variable for each process, use an interprocess variable instead of a process variable. 4D only creates one instance of the interprocess variable table; this instance is shared by all processes.

Note: The interprocess variable table is limited to a total of 32k.

To control ÒFor Loops,Ó use Long Integer variables instead of Reals

Using long integer numerical counters in your For Loops optimizes your application in two ways:

- ¥ A long integer is only 4 bytes, while a Real variable is 10 bytes.

¥ Performing a calculation on a Real value takes several times longer than performing the same calculation on a Long Integer value. This is true even for a simple calculation such as $2 + 1 = 3$.

Clear variable-length variables when they are no longer needed

When we mention "variable-length variables," we include variables of type Text, Picture, Array, and BLOB. You can clear all of these variables with the command:

```
CLEAR VARIABLE (variable)
```

In interpreted mode, `CLEAR VARIABLE` erases the variable from memory. In compiled mode, `CLEAR VARIABLE` only resets the variable to its default type value (i.e., empty string for String and Text variables, 0 for numeric variables, no elements for arrays, etc.). The variable still exists, but it takes a minimum amount of memory.

Declare strings and string arrays to odd lengths

When you declare the length of a string, 4D adds a one-byte header to store the length of the string. For example, when you declare a string as `C_STRING (10;sMyString)`, the actual length of that string is 11. Also, 4D follows the Macintosh convention for handling strings, which means that every string must be an even number of bytes in length. This means that when you declare a string as 10 characters, it will actually be 12 characters in size. The same is true with strings in an array: when you declare an array as `ARRAY STRING (10;asMyStrArray;50)`, you will actually have a 50-element array with each element 12 bytes in length. In each element, you have a wasted byte that you cannot use. In a 50-element string array, you are wasting 50 bytes. For this reason, you should declare your strings and string arrays to odd lengths. For example:

```
C_STRING (11;sMyString)
ARRAY STRING (11;asMyStrArray;50)
```

Size variable-length variables at the maximum size you think you will need

Each time you increase the size of a variable-length object, you might be causing a BlocMove in memory. Each time 4D has to move the object to a new location, this creates two problems: finding a place in memory to put the object, and further fragmentation of the memory. Therefore, instead of declaring an object at a small size and making it larger each time you need to add data, you should initially declare the object, and then immediately resize it to the largest size you think you will need. Then, after you have filled the object with data, you can resize it to the exact size that you need. Here are some examples of this technique:

Sizing a text variable:

```
C_LONGINT($counter)
C_TEXT($tText)
$tText:=1000 * Char(32) `Text variable resized to 1000 characters.
```

Sizing an Array:

```
ARRAY STRING(15;asStringArray;0)
ARRAY STRING(15;asStringArray;50) `Resize the array to 50 elements
`Perform your operations to fill the array, you fill 27 elements
ARRAY STRING(15;asStringArray;27) `Resize the array to 27 elements
```

Sizing a BLOB:

```
C_BLOB(blMyBlob)
SET BLOB SIZE (blMyBlob;1000) `Resize the BLOB to 1000 bytes
`Perform your operations to fill the BLOB, you fill 877 bytes.
SET BLOB SIZE(blMyBlob;877) `Resize the BLOB to 877 bytes.
```

Sizing a Picture:

```
C_PICTURE (pMyPictVar)
pMyPictVar:=[Pictures_Objects]Picture `Copy this field into the picture variable.
```

```
pMyPictVar:=pMyPictVar * 10 `Set the variable to 10 times its original size.  
`Perform your picture arithmetic operations to fill the picture
```

Whenever possible, pass a pointer to a variable length object, rather than passing the object itself

When you pass a parameter to a method, 4D must make a copy of that value and put it into the appropriate local variable inside the method. The first passed parameter is copied into \$1, the second passed parameter is copied into \$2, and so on. For example, suppose you write this line of code:

```
ProcessPicture ([Picture_Objects]PictureField)
```

The passed parameter is a 500k Picture field, [Picture_Objects]PictureField. While the method ProcessPicture is running, you have two copies of that picture in memory: one copy in the picture field and one copy in the variable \$1. That picture is using a total of one megabyte in memory. Instead, pass a pointer to [Picture_Objects]PictureField in this manner:

```
ProcessPicture (->[Picture_Objects]PictureField)
```

The size of the pointer is 4 bytes, not 500k, so you have saved almost 500k simply by passing a pointer to the field instead of passing the field itself. Inside the method, when you need to use the value represented by the pointer, you can just de-reference the pointer. For example:

```
C_LONGINT($Size)  
$Size:=Picture Size ($1->)
```

Note: We said that you should pass a pointer "whenever possible" because sometimes it is not possible or not advisable to pass a pointer. For example, you are not allowed to pass a pointer as a parameter to any method that creates a new process. You may not pass a pointer to the `New Process` function or to the `Execute On Server` function. Also, if you will be referring to the object several times within a loop, keep in mind that the loop will be a bit slower because each time through the loop, 4D must de-reference the pointer in order to read the actual value of the object.

Be aware of the memory penalties of automatic relations several levels deep

If you have set up automatic relations between two files, there are certain situations when 4D automatically creates related many selections and, for each related Many table, loads the first record in the selection. When 4D does this automatic loading, it does so for the first level of automatic one to many relation and for up to five levels of many to one relations. For example, if the automatic relations are five levels deep, 4D will load the selection and records for the first level. If you will not need the related many selection and record, this can be a waste of memory. Especially if the loaded record contain large pictures, text fields, or BLOBs.

The following commands automatically load the related many selections and records:

```
ADD RECORD  
MODIFY RECORD  
MODIFY SELECTION (in data entry)  
QUERY BY FORMULA  
QUERY SELECTION  
QUERY  
PRINT SELECTION
```

To avoid this waste of memory, you can set your one-to-many relations to Manual in the Design environment, and then call the command `AUTOMATIC RELATIONS (True;True)` when you need to make the one-to-many relations Automatic.

Note: If you set the relations to Automatic in the Design environment, you cannot override that setting in code. You can go from Manual to Automatic, but you cannot go from Automatic to Manual in the code.

Launch the most frequently used processes on startup, and hide those processes when not in use. The process stack needs special consideration because when a process is running, the stack cannot be relocated nor purged. This means that you

can badly fragment the memory by starting and stopping processes. For this reason, launch the most frequently used processes on startup, and then simply pause and hide those processes when not in use.

Here is an example of three lines of code in your On Startup method:

```
<>InvoiceProcess:=New Process (ÒInvoiceMethodÓ;32000;ÓInvoicing ProcessÓ)  
HIDE PROCESS (ÒInvoicing ProcessÓ)  
PAUSE PROCESS (ÒInvoicing ProcessÓ)
```

Note: It is a good programming habit to hide a process before pausing it. If you pause a process without hiding it, the user can become confused, for example, when clicking on a visible window and nothing happens.

Summary

At any given time, the performance and stability of any 4D application is affected by the amount of RAM that is available for loading objects into memory, and by the sizes of the blocks of available RAM. In Part 1 of this technical note, you learned how 4D uses memory, and what you can do in your 4D forms in order to keep the memory Òclean.Ó In Part 2, you learned how to write 4D code that helps to keep the memory clean.