

DLL Wizard

*Reference Guide
for Windows*



DLL Wizard Reference Guide

Windows® version

*Copyright © 1997-2000 4D SA.
All rights reserved.*

The Software described in this manual is governed by the grant of license in the 4D Product Line License Agreement provided with the Software in this package. The Software, this manual, and all documentation included with the Software are copyrighted and may not be reproduced in whole or in part except for in accordance with the 4D Product Line License Agreement.

4th Dimension, 4D, 4D Insider, the 4D logo, 4D Server, 4D, and the 4D logo are registered trademarks of 4D SA.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Apple, Macintosh, Mac, Power Macintosh, LaserWriter, ImageWriter, ResEdit, and QuickTime are trademarks or registered trademarks of Apple Computer, Inc.

All other referenced trade names are trademarks or registered trademarks of their respective holders.

Contents

Introduction	5
Chapter 1	Installation 7
	Installation on disk 7
	Limits 7
Chapter 2	Getting started 9
	Basic steps 9
	Note for DLL Tools users 9
Chapter 3	Samples 11
	Learning DLL Wizard through examples 11
	Handling a structure. 15
Chapter 4	DLLs and functions 17
	DLL name 17
	Procedure declaration 17
	Calling convention 18
	Parameters and 4D variables types 18
	Parameters 18
	4D Variable types 19
Chapter 5	WinMem Plug-In 21
	WinMem: Allocation 21
	WM GlobalAlloc 21
	WM GlobalFree 22
	WM GlobalLock 22
	WM GlobalUnlock 23
	WM GlobalSize 23
	WM GlobalReAlloc 24
	WinMem: Access 24
	WM SET BYTE 24

WM SET SHORT25

WM SET LONG.....25

WM SET DOUBLE.....26

WM SET FLOAT26

WM SET CSTRING26

WM SET PSTRING27

WM SET STRING27

WM SET BLOB28

WM Get byte28

WM Get short.....28

WM Get long29

WM Get double29

WM Get float29

WM Get cstring30

WM Get pstring30

WM Get string31

WM Get BLOB31

Introduction

4th Dimension offers a robust procedural language containing hundreds of commands. However, developers may need to access specific functions which are not included in the basic 4D language.

Often, the required functionality already exists as a Dynamic Link Library (DLL). A DLL is a piece of code that can be called from any application and is shared between different applications. The link is dynamic because any function located in the DLL can be called simply by knowing its name.

The Windows programming interface itself is based on DLLs, and most of the functions are located in “Kernel32.DLL”, “User32.DLL” or “GDI32.DLL”.

One way to call a Windows function is to write a 4th Dimension extension in C or C++ that will call the specific function. This is recommended for complex tasks, but if you need to call a simple routine (for example, calling the Windows API function “GetDiskFreeSpace” to know the remaining space on your hard disk) writing a 4D extension to perform this operation may not be worth it.

In order to use DLL Wizard, you will need:

- A native development environment installed on your computer,
- Knowledge of C or C++,
- To know how to write and compile a 4D extension...

DLL Wizard provides an easy way to bypass this problem. With DLL Wizard, you can declare and call any function located in a DLL via a graphic interface. You can pass parameters and get values. To do so, DLL Wizard generates a plug-in that makes these functions available in 4D.

DLL Wizard can also provide useful utilities that can allocate memory blocks and read or write values in these blocks (they are implemented in a package named WinMem).

DLL Wizard replaces the DLL Tools plug-in.

1

Installation

Installation on disk

DLL Wizard is only available for Windows platforms running 4D version 6.5 or later. DLL Wizard requires a minimum of 2 MB of disk space.

- To install DLL Wizard on your hard disk:

1 Copy DLL Wizard.4DB (or .4DC) and DLL Wizard.4DD on your disk.

It is ready to use.

This 4D database is designed to work on Windows. The plug-ins generated with DLL Wizard can work on 4D v6 Windows version.

Limits

DLL Wizard handles only 32-bit DLLs.

Maximum supported values are:

- Number of designed plug-ins: limited to disk space;
- Procedures in a plug-in: thousands¹;
- Parameters per procedure: 25;
- DLLs called in a plug-in: 255*.

1. The real values may be lower because of the configuration used (operating system, available memory and disk space).

2

Getting started

Basic steps

This section describes how to take your first steps with DLL Wizard.

- 1 **Open DLL Wizard, then choose *New...* in the *File* menu.**
 - On the first page (plug-in), name the plug-in.
 - On the second page (procedures), add the function declarations.
- 2 **Click on the button *Add a procedure* (see *Samples* section) to do so.**
- 3 **On the third page (final step), select the database for which the plug-in is designed. DLL Wizard will then create the plug-in in the WIN4DX folder located next to the database.**

If necessary the WIN4DX folder will be created.

You can also install the WinMem plug-in if you plan to make memory allocations.

If you wish to modify a plug-in, choose **Open...** in the **File** menu.

Note for DLL Tools users

Most of your work in 4D will be made in DLL Wizard, your methods will only contain DLL calls, not call declarations.

Unlike DLL Tools, you do not have to load or free a library:

- The needed libraries are automatically loaded at start-up;
- The libraries remain loaded as long as the database is opened;
- The libraries are freed at exit.

The function declarations are made in DLL Wizard (its main goal), you simply have to call them in 4D's methods (you do not need to call LoadLibrary). You do not need the 4D command EXECUTE to perform calls to a DLL in a compiled database. Constants can also be given as parameters.

Memory allocation and access functions are provided in a separated package.

What was written with DLL Tools:

```
Lib:=LoadLibrary("TheDLL.dll")  
Call:=DLL declare(Lib;"BOOL TheProcedure(DWORD,DWORD)";0)  
Value2:=20  
Err:=CallDLL(Call;Value1;Value2;ReturnValue)  
FreeCall(Call)  
FreeLibrary(Lib)
```

Is now reduced to:

```
ReturnValue:=TheProcedure(Value1;20)
```

3

Samples

Learning DLL Wizard through examples

The following example allows you to retrieve the command given to 4th Dimension at startup using the Windows API function `GetCommandLine`. Looking at this function with Quick View gives us a `GetCommandLineA` and a `GetCommandLineW` function.

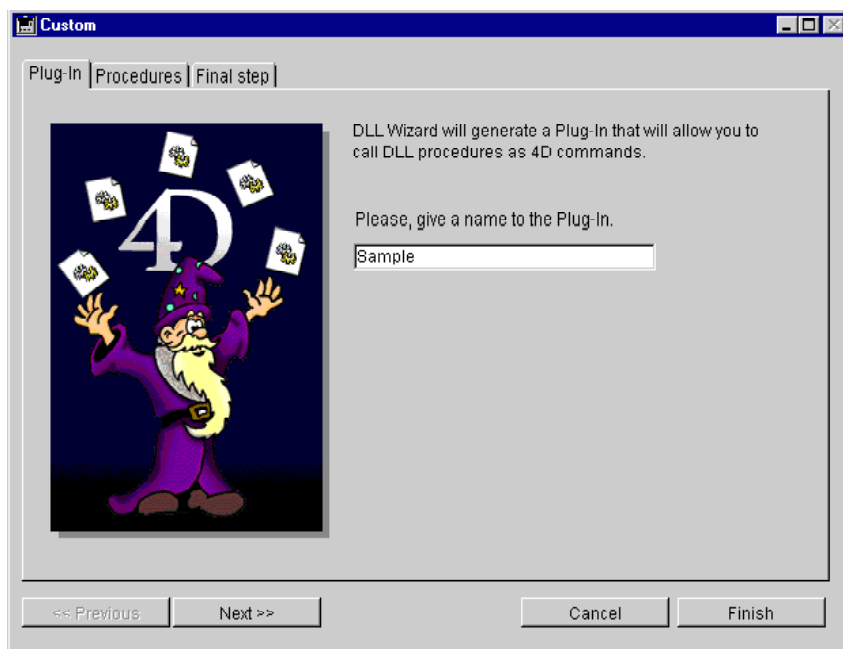
This is because `Kernel32.dll` implements two functions: one receiving a standard ASCII string (`GetCommandLineA`), and the other receiving a wide-character string, (`GetCommandLineW`). When you call `GetCommandLine` from a C or C++ source code, the linker knows if you work with ASCII or Wide-Character strings and calls the appropriate function in the DLL. With DLL Wizard, you directly call the DLL, and you have to know which function to call. In our case, 4D works internally with ASCII code, so we will use the `GetCommandLineA`.

Visual C++ gives the following information about this function:

“The **`GetCommandLine`** function returns a pointer to the command-line string for the current process.”

`LPTSTR GetCommandLine(VOID)`

- 1 Select *New* in the *File* menu.
- 2 In the Plug-in tab, name the plug-in (*Sample*).

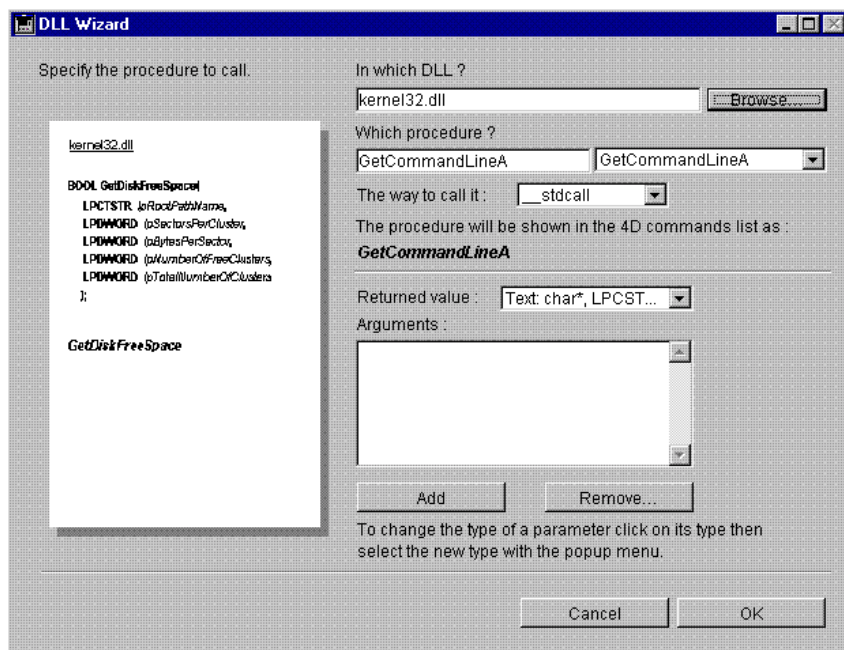


- 3 In the Procedures tab, click on *Add a procedure*.



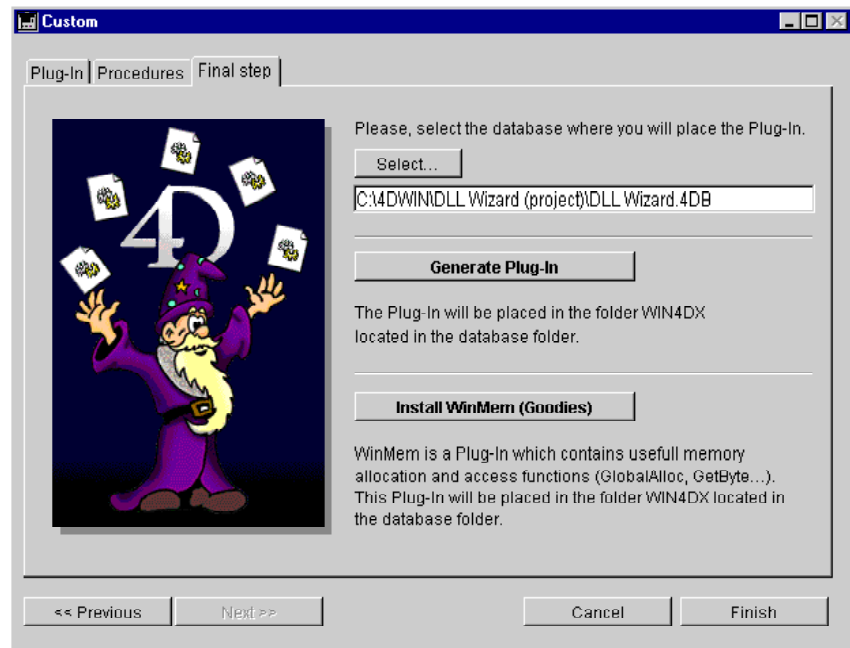
The routine entry form allows you to set the parameters and returned value of a routine:

- Name the DLL: **kernel32.dll**;
- Name the routine: **GetCommandLineA**;
- Select the calling convention: **__stdcall**;
- Select the returned value type: **text**;

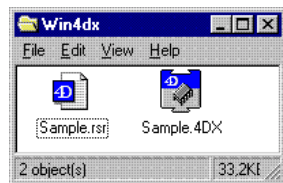


4 Validate the modifications.

5 In the Final step window, select the database for which the plug-in is designed.

6 Click on *Generate Plug-In*.

Look in the WIN4DX folder (located in the specified database folder), you will find the plug-in Sample.4DX. Open the database, the plug-in is ready to be used.



In a 4D method, the procedure will be called as follow:

```
$cmdline:=GetCommandLineA
```

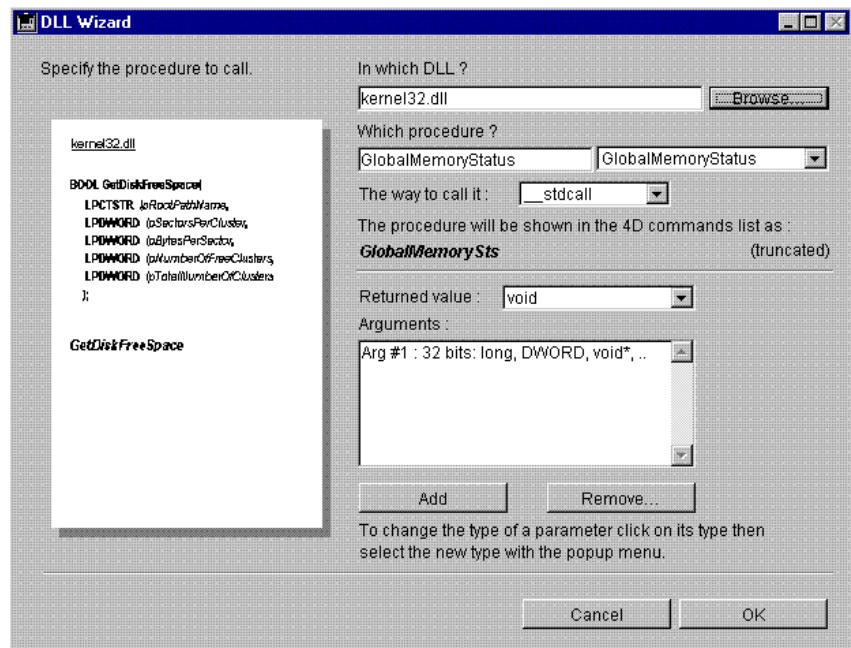
Handling a structure

We plan to use GlobalMemoryStatus whose definition (in Visual C++) is:

```
VOID GlobalMemoryStatus(
    LPMEMORYSTATUS    lpBuffer    // pointer to the memory status structure
);

typedef _MEMORYSTATUS {
    DWORD    dwLength;                // sizeof(MEMORYSTATUS)
    DWORD    dwMemoryLoad;            // percent of memory in use
    DWORD    dwTotalPhys;             // bytes of physical memory
    DWORD    dwAvailPhys;             // free physical memory bytes
    DWORD    dwTotalPageFile;         // bytes of paging file
    DWORD    dwAvailPageFile;         // free bytes of paging file
    DWORD    dwTotalVirtual;          // user bytes of address space
    DWORD    dwAvailVirtual;          // free user bytes
} MEMORYSTATUS, *LMEMORYSTATUS;
```

In the Procedure entry form you will declare it as follows:



Note As the procedure name (GlobalMemoryStatus) is over 15 characters, it is truncated (the first 14 characters plus the last character).

- ▼ The call will be performed in a 4D method as follows:

- ` We allocate a buffer for the structure
- ` we assume that WinMem plug-in is installed

```
$lpBuffer:=WM GlobalAlloc (GMEM_FIXED;32)  
WM SET LONG ($lpBuffer;32) ` we indicate the size of the buffer  
` (see MEMORYSTATUS definition)
```

- ` We perform the call

```
GlobalMemorySts ($lpBuffer)
```

- ` We retrieve the information
- ` specifying the address and the offset
- ` of the members

```
$dwMemoryLoad:=WM Get Long ($lpBuffer+4)  
$dwTotalPhys:=WM Get Long ($lpBuffer+8)  
$dwAvailPhys:=WM Get Long ($lpBuffer+12)  
$dwTotalPageFile:=WM Get Long ($lpBuffer+16)  
$dwAvailPageFile:=WM Get Long ($lpBuffer+20)  
$dwTotalVirtual:=WM Get Long ($lpBuffer+24)  
$dwAvailVirtual:=WM Get Long ($lpBuffer+28)
```

```
$err:=WM GlobalFree ($lpBuffer)
```


4

DLLs and functions

DLL name

If you specify only the DLL name, the DLL must be located either in the application directory, in the current directory or in the system directory.

If you specify a full path name, the DLL needs to be located in the specified folder.

When the database is opened, if a DLL is not found, you will see an alert “*XXX.dll* not loaded”. Normally this should not happen with DLLs such as *Kernel32.dll*, *User32.dll*, etc.

Procedure declaration

To declare a procedure, you need to know the number, the types of parameters and the returned value.

To do that, you need a description of the Windows 32-bit API. You can find this information either in a Windows programming book or in the on-line help of a development system, like Visual or Borland C++. If you are using a specific DLL (for example, fax software), you can have a look in the documentation or in the header files.

When the database is opened, if a procedure is not found in the DLL, you will get an alert “*proc_name* not found”.

If you do not know in which DLL the procedure is located, you can use Quick View or Dumpbin, or select the DLL clicking on Browse (see sample above).

Calling convention

In most cases, a procedure is called with Pascal conventions (`__stdcall`).

If your DLL uses C conventions, select `__cdecl`.

Parameters and 4D variables types

Parameters

You can declare the following parameter types:

- **8-bit:**
char, BOOL, BYTE, CHAR, UCHAR, BOOLEAN, CCHAR
- **16-bit:**
short, WORD, UWORD, SHORT, USHORT
- **32-bit:**
long, word, int, short*, word*, long*, int*, void*, DWORD, LONG, LPVOID, UINT, GLOBALHANDLE, HANDLE, HLOCAL, LPDWORD, LPBOOL, LPBYTE, LPWORD, LPLONG
- **string pointers:**
char*, LPCSTR, LPSTR, LPCTSTR, LPTSTR, NPSTR, PCSTR, PCWSTR, PSTR, PTSTR
- **double values (64 bits):**
double, GLdouble
- **float values (32 bits):**
float
- **void return:**
void, VOID

If your function accepts a parameter type that is not on this list, you should pass a compatible type parameter. If your function waits for a pointer to a structure, you need to declare it as a 32 bit pointer, such as `void*`.

Particular case: Structure parameters

If you want to declare a structure as a parameter in the function(not a pointer to the structure, but the structure content), you will have to successively pass all the members of the structure.

- ▼ Example: Calling a function that accepts a RECT parameter.

Original declaration:

```
void DoSomething( RECT theRect );
```

RECT is declared in the Windows API as a structure containing 4 longs. You need to declare your call as follow:

Arg #1: long

Arg #2: long

Arg #3: long

Arg #4: long

You can pass either 4D variables or constants. DLL Wizard can automatically handle some conversions for you, like converting a real to a long.

4D Variable types

Here are the possible 4D variable types you can pass for each declared type :

- **8-bit: (char, BOOL, BYTE, CHAR, UCHAR, BOOLEAN, CCHAR)**

You can pass an integer, longint, or real 4D variable. The most appropriate variable type is Integer or Longint. Passing a 4D Real variable is possible but slower (a conversion will be performed).

If your function is expecting a signed byte, the example of the SetByte function will show you how to convert an unsigned byte into a signed byte.

- **16-bit: (short, WORD, UWORD, SHORT, USHORT)**

You can pass an integer, longint, or real 4D variable. The most appropriate variable type is a 4D integer variable. Passing a 4D Real variable value is possible but slower (a conversion will be performed).

- **32-bit: (long, word, int, void*, DWORD, LONG...)**

You can pass an integer, longint, or real 4D variable. The most appropriate variable type is a 4D longint variable. Passing a 4D Real variable value is possible but slower (a conversion will be performed).

- **string pointers: (char*, LPCSTR, LPSTR, LPCTSTR, LPTSTR...)**

You can pass an alpha or a text variable. If you pass an alpha variable, be sure that the DLL function will return you a string of an appropriate size. If you declare an alpha variable of 20 chars long, and you call a DLL function that will return a string of 80 characters long, you will only get the 20 first characters in your 4D variables. If your DLL is returning larger strings, you can use a text variable. The limitation for text variable is 32000 characters.

Because 4D internally uses Macintosh extended characters, the strings passed to the DLL functions will be automatically converted to ANSI before calling the function, and reconverted into Macintosh ASCII code afterwards.

- **float values (32 bits): (float)**

You can pass an integer, longint, or real 4D variable. The most appropriate variable type is a 4D real variable. Passing an other numeric value is possible but slower (a conversion will be performed). Loss of precision may occur during conversions because 4D real variable are double.

- **double values (64 bits): (double, GLdouble)**

You can pass an integer, longint, or real 4D variable. The most appropriate variable type is a 4D real variable. Passing an other numeric value is possible but slower (a conversion will be performed).

5

WinMem Plug-In

This plug-in provided by DLL Wizard contains useful functions that allocate global handles or pointers.

These functions are designed to give you easy access to basic memory management routines for Windows.

WinMem plug-in also contains functions that will allow you to read or write data inside the allocated blocks (see section “[WinMem: Access](#)”, page 24).

WinMem: Allocation

WM GlobalAlloc

WM GlobalAlloc (Flags; Size) → Long

Parameters	Type	Description
<i>Flags</i>	LongExpr	Object allocation attributes
<i>Size</i>	LongExpr	Number of bytes to allocate

This command is mapped on the Windows API GlobalAlloc function. It allocates the requested amount of memory and returns a handle on the allocated block.

Some functions declared with DLL Wizard will request a pointer to a memory block as a parameter. You can create this block with GlobalAlloc.

The following values may be used, or added together:

- 0 GMEM_FIXED, allocates fixed memory. The returned value is a pointer to the memory block. You do not need to lock or unlock the block before using it.
- 2 GMEM_MOVEABLE, allocates movable memory. The returned value is a handle of the memory object. You need to call GlobalLock to get a pointer to the memory block.
- 8192 GMEM_DDESHARE, the allocated block may be used for a DDE conversation, or clipboard operations.
- 256 GMEM_DISCARDABLE, the allocated block can be discarded. This flag can not be combined with the GMEM_FIXED flag.
- 16 GMEM_NOCOMPACT, does not compact memory to satisfy allocation request.
- 32 GMEM_NODISCARD, does not discard memory to satisfy allocation request.
- 64 GMEM_ZEROINIT, initializes memory content to zero.

If the function fails, the returned value is zero.

WM GlobalFree

WM GlobalFree (Handle) → Long

Parameters	Type	Description
<i>Handle</i>	LongExpr	Handle to the global memory object

This function frees the memory block allocated by the **GlobalAlloc** functions.

If the function succeeds, the returned value is 0. If the function fails, the return value is the handle of the global memory object.

WM GlobalLock

WM GlobalLock (Handle) → Long

Parameters	Type	Description
<i>Handle</i>	LongExpr	Handle to the global memory object

This function locks the global memory object and returns a pointer to the first byte of the object memory block. Once locked, the memory block will not move or be discarded until you unlock it.

You need to call this function only for objects allocated using the `GMEM_MOVEABLE` flag.

There is an internal counter that is increased each time you lock the handle, and decreased each time you unlock it. The block will actually be unlocked only when the counter is set to zero. This is useful when a procedure A, that locks the handle, calls a procedure B, that locks and unlocks the same handle, and is returned to procedure A with the handle still locked.

WM GlobalUnlock

WM GlobalUnlock (Handle) → Long

Parameters	Type	Description
<i>Handle</i>	LongExpr	Handle to the global memory object

This function unlocks the global memory object. Once unlocked, the pointer on the memory block is no longer valid, and using it may cause an access privilege exception.

You need to call this function only for objects allocated using the `GMEM_MOVEABLE` flag.

The handle will actually be unlocked only if the internal lock counter is set to zero. If the object is still locked after this call (if the lock counter has not fallen to zero) the returned value is 1, and 0 if the object is unlocked.

WM GlobalSize

WM GlobalSize (Handle) → Long

Parameters	Type	Description
<i>Handle</i>	LongExpr	Handle to the global memory object

This function returns the current size, in bytes, of the specified memory object.

If the specified handle is no longer valid, or if the object has been discarded, the returned value will be zero.

Note The size of the memory block may be larger than the size requested when the memory was allocated, so you cannot rely on this size to compute, for example, the number of elements stored in the block.

WM GlobalReAlloc

WM GlobalReAlloc (Handle; Size; Flags) → Long

Parameters	Type	Description
<i>Handle</i>	LongExpr	Handle to the global memory object
<i>Size</i>	LongExpr	New size of the block
<i>Flags</i>	LongExpr	How to reallocate object

This function changes the size or attributes of the specified memory object.

If the specified handle is no longer valid, or if the object has been discarded, the returned value will be zero.

Note The size of the memory block may be larger than the size requested when the memory was allocated, so you cannot rely on this size to compute, for example, the number of elements stored in the block.

WinMem: Access

These functions allow you to put specific values anywhere in the memory. This is useful if a DLL function needs a pointer to a structure containing particular values. With the memory management functions, you can allocate a memory block and fill the block using these routines.

Note When dealing with a structure, do not forget to respect the byte alignment.

WM SET BYTE

WM SET BYTE (Pointer; Value)

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address
<i>Value</i>	LongExpr	Value of the byte to set (0 -> 255)

This command sets an unsigned byte located at the specified address. An unsigned byte is one byte long (8 bits) and can hold a value between 0 and 255.

- ▼ If you want to pass a signed value between -128 and +127, you can convert your value before passing it to the routine using this conversion algorithm:

```

If ( value<0)
    value:=256+value
End if
WM SET BYTE(pt;value)

```

WM SET SHORT

WM SET SHORT (Pointer; Value)

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address
<i>Value</i>	LongExpr	Value of the short to set (0->65535)

This command sets an unsigned short value at the specified address. An unsigned short is 2 bytes long (16 bits) and can hold a value between 0 and 65535.

- ▼ If you want to pass a signed value between -32768 and +32767, you can convert your value to a signed short before passing it to the routine using this conversion algorithm:

```

If ( value<0)
    value:=65536+value
End if
WM SET SHORT(pt;value)

```

WM SET LONG

WM SET LONG (Pointer; Value)

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address
<i>Value</i>	LongExpr	Signed long value to set

This command sets a long value at the specified address. A long value is 4 bytes long (32 bits) and can hold a value between -2147483648 and +2147483647.

WM SET DOUBLE

WM SET DOUBLE (Pointer; Value)

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address
<i>Value</i>	RealExpr	Double value to set

This command sets a double value at the specified address. A double value is 8 bytes long (64 bits) and can hold a real value. A real value stored on 64 bits may be positive or negative and ranges from 1.7 E-308 to 1.7 E+308 with 15 significant digits.

▼ Example:

WM SET DOUBLE(pt; 3,5667)**WM SET FLOAT**

WM SET FLOAT (Pointer; Value)

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address
<i>Value</i>	RealExpr	Float value to set

This command sets a float value at the specified address. A float value is 4 bytes long (32 bits) and can hold a real value. A real value stored on 32 bits may be positive or negative and range from 3.4 E-38 to 3.4 E+38 with 9 significant digits. Loss of precision may occur during conversions because 4D real variables are double.

▼ Example:

WM SET FLOAT(pt; 3.5667)**WM SET CSTRING**

WM SET CSTRING (Pointer; Text)

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address
<i>Text</i>	TextExpr	Text to copy

This command will copy the specified text at the specified address.

This text will be converted into ANSI characters (4D internally uses the Mac ASCII code representation). A null character will be added at the end of the text, because most DLL functions are expecting null terminated strings.

If your function is waiting for an UNICODE string, you will first need to call a DLL function that handles this conversion, such as WideCharToMultiByte, located in Kernel32.DLL.

▼ Example:

WM SET CSTRING(pt; "Hello World")

WM SET PSTRING

WM SET PSTRING (Pointer; Text)

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address
<i>Text</i>	TextExpr	Text to copy

This command will copy the specified text at the specified address.

This text will be converted into ANSI characters (4D internally uses the Mac ASCII code representation). A length character will be added at the beginning of the text. If your function is waiting for an UNICODE string, you will first need to call a DLL function that handles this conversion, such as WideCharToMultiByte, located in Kernel32.DLL.

▼ Example:

WM SET PSTRING (pt; "Hello World")

WM SET STRING

WM SET STRING (Pointer; Text)

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address
<i>Text</i>	TextExpr	Text to copy

This command will copy at the specified address the specified text.

This text will be converted into ANSI characters (4D internally uses the Mac Ascii code representation). A length character will be added at the beginning of the text. If your function is waiting for an UNICODE string, you will first need to call a DLL function that handles this conversion, such as WideCharToMultiByte, located in Kernel32.DLL.

▼ Example:

WM SET STRING (pt; "Hello World")

WM SET BLOB

WM SET BLOB (Pointer; Value; Length)

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address
<i>Value</i>	BlobExpr	Bytes to copy
<i>Length</i>	LongExpr	Number of bytes to set

This command will copy the given number of bytes of the BLOB at the specified address. Data is copied from offset 0 of the BLOB.

WM SET BLOB(pt;vBlob;200)

These functions help you to read specific values anywhere in memory. They are useful if a DLL function returns a pointer to a structure or data containing particular values that you need to read.

WM Get byte

WM Get byte (Pointer) → Value

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address

This command reads an unsigned byte located at the specified address. An unsigned byte is one byte long (8 bits) and it can hold a value from 0 to 255.

- ▼ If you want to read a signed value, from -128 to +127, you can convert your value after reading it, using this conversion algorithm:

```
value:=WM Get byte(pt)
If ( value>127)
    value:=value-256
End if
```

WM Get short

WM Get short (Pointer) → Value

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address

This command reads an unsigned short value at the specified address. An unsigned short is 2 bytes long (16 bits) and it can hold a value in the range 0 to 65535.

- ▼ If you want to read a signed value, from -32768 to +32767, you can convert your value to a signed short after reading it, using this conversion algorithm:

```
value:=WM Get short(pt;value)
If (value>32767)
    value:=value-65536
End if
```

WM Get long

WM Get long (Pointer) → Value

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address

This command reads a long value at the specified address. A long value is 4 bytes long (32 bits) and can hold a value from -2147483648 to +2147483648.

WM Get double

WM Get double (Pointer) → Value

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address

This command reads a double value at the specified address. A double value is 8 bytes long (64 bits) and can hold a real value. A real value stored on 64 bits may be positive or negative and ranges from 1.7 E-308 to 1.7 E+308 with 15 significant digits.

- ▼ Example:

```
C_REAL(dbl)
dbl:=WM Get double(pt)
```

WM Get float

WM Get float (Pointer) → Value

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address

This command reads a float value at the specified address. A float value is 4 bytes long (32 bits) and can hold a real value. A real value stored on 32 bits may be positive or negative and ranges from 3.4 E-38 to 3.4 E+38 with 9 significant digits. Loss of precision may occur during conversions because 4D real variables are double.

▼ Example:

```
C_REAL(dbl)
dbl:=WM Get float(pt)
```

WM Get cstring

WM Get cstring (Pointer) → Text

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address

This command will return a copy of the text stored at the specified address, assuming it is a C string. This text will be converted from ANSI to Macintosh characters (4D internally uses the Mac ASCII code representation).

If you want to read a UNICODE string, you must first call a DLL function to handle the conversion, like MultiByteToWideChar, located in Kernel32.DLL.

▼ Example:

```
C_TEXT(vText)
vText:=WM Get cstring(pt)
```

WM Get pstring

WM Get pstring (Pointer) → Text

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address

This command will return a copy of the text stored at the specified address, assuming it is a Pascal string. This text will be converted from ANSI to Macintosh characters (4D internally uses the Mac ASCII code representation).

If you want to read a UNICODE string, you must first call a DLL function to handle the conversion, like MultiByteToWideChar, located in Kernel32.DLL.

▼ Example:

```
C_TEXT(vText)
vText:=WM Get pstring(pt)
```

WM Get string

WM Get string (Pointer; Length) → Text

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address
<i>Length</i>	IntegerExpr	Number of bytes to copy

This command will return a copy the given number of bytes of the text stored at the specified address. This text will be converted from ANSI to Macintosh characters (4D uses internally the Mac ASCII code representation).

If you want to read an UNICODE string, you must first call a DLL function to handle the conversion, such as MultiByteToWideChar, located in Kernel32.DLL.

▼ Example:

C_TEXT(vText)
vText:=**WM Get string**(pt;len)

WM Get BLOB

WM Get BLOB (Pointer; Length) → Blob

Parameters	Type	Description
<i>Pointer</i>	LongExpr	Memory Address
<i>Length</i>	LongExpr	Number of bytes to get

This command will return a BLOB which contains a copy of the given number of bytes at the specified address.

▼ Example:

C_BLOB(vBlob)
vBlob:=**WM Get BLOB**(pt;200)

