DLL Wizard 1.0.2

Windows® Version

Documentation

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 generic 4D language.

Often, the required functionality already exists as a Dynamic Link Library (DLL). A DLL is piece of code which can be called from any application; it is shared between different applications, and 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, 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++, which will call the specific function.
This is an appropriate solution if your extension performs a complex task. But, if you just 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.

You need:
- a native development environment installed on your computer,
- to know C or C++,
- to write and compile a 4D extension...

The DLL Wizard gives you an easy way to bypass this problem.

With DLL Wizard, you can declare any function located in a DLL via a graphic interface, and call it. You can pass parameters and get returned values. To do so DLL Wizard will generate a plug-in to make these functions available in 4D.
DLL Wizard can also provide useful utilities to allocate memory blocks and to read or write values in these blocks (they are implemented in a package named WinMem).

DLL Wizard replaces the DLL Tools Plug-In.

Installation

DLL Wizard is available only for 4th Dimension for Windows. It requires 4D version 6.0 or later. DLL Wizard requires for itself a minimum of 2 Mb of disk space.

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 95 or Windows NT. The plug-ins generated with DLL Wizard can work on both 4D v6 and v3.5, on Windows 95, Windows NT and Windows 3.11.

Limits

DLL Wizard handles only 32 bits DLLs.

Absolute maxima are:
- number or designed plug-ins: limited to disk space;
- procedures in a plug-in: thousands *;
- parameters per procedure: 25;
- DLLs called in a plug-in: 255 *.

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

Getting started

Open DLL Wizard, then choose New ... in the File menu.

On the first page (plug-in), you have to name the plug-in.

On the second page (procedures), you add the function declarations.
To do so, click on the button Add a procedure (see Samples).

On the third page (final step), you select the database for which the plug-in is designed, DLL Wizard will 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 WinMem plug-in if you plan to make memory allocations.

When you want to modify a plug-in, choose Open... in the File menu.

Note for DLL Tools users

Most of the work you were doing in 4D will be made in DLL Wizard, your methods will only contain calls to DLLs, not the call declarations.

Unlike DLL Tools you do not have to load and free a library:
- the needed libraries are automatically loaded at start-up;
- they remain loaded as long as the database is opened.
- they are released at exit ;

The functions declarations are made in DLL Wizard (its main goal), you just have to call them in 4D methods (You do not need to call LoadLibray, ...). 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 you wrote 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)
```

Samples

Getting started:

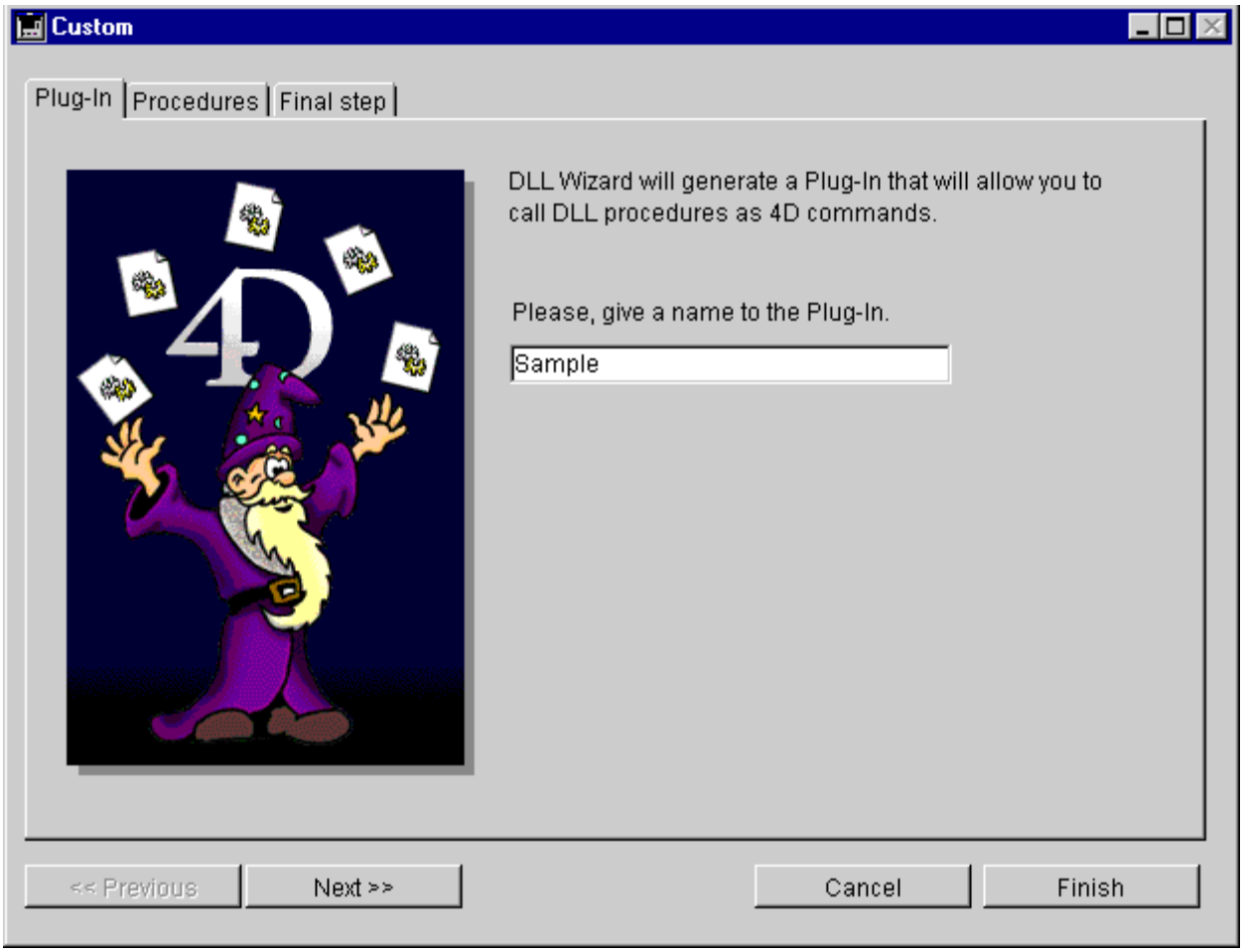
The following example allows you to retrieve the command given to 4th Dimension at startup. The function `GetCommandLine` of Windows API will perform that. Looking at this function Quick View gives us a `GetCommandLineA` and a `GetCommandLineW` function. This is because Kernel32.dll implements two functions, 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 choose to 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:

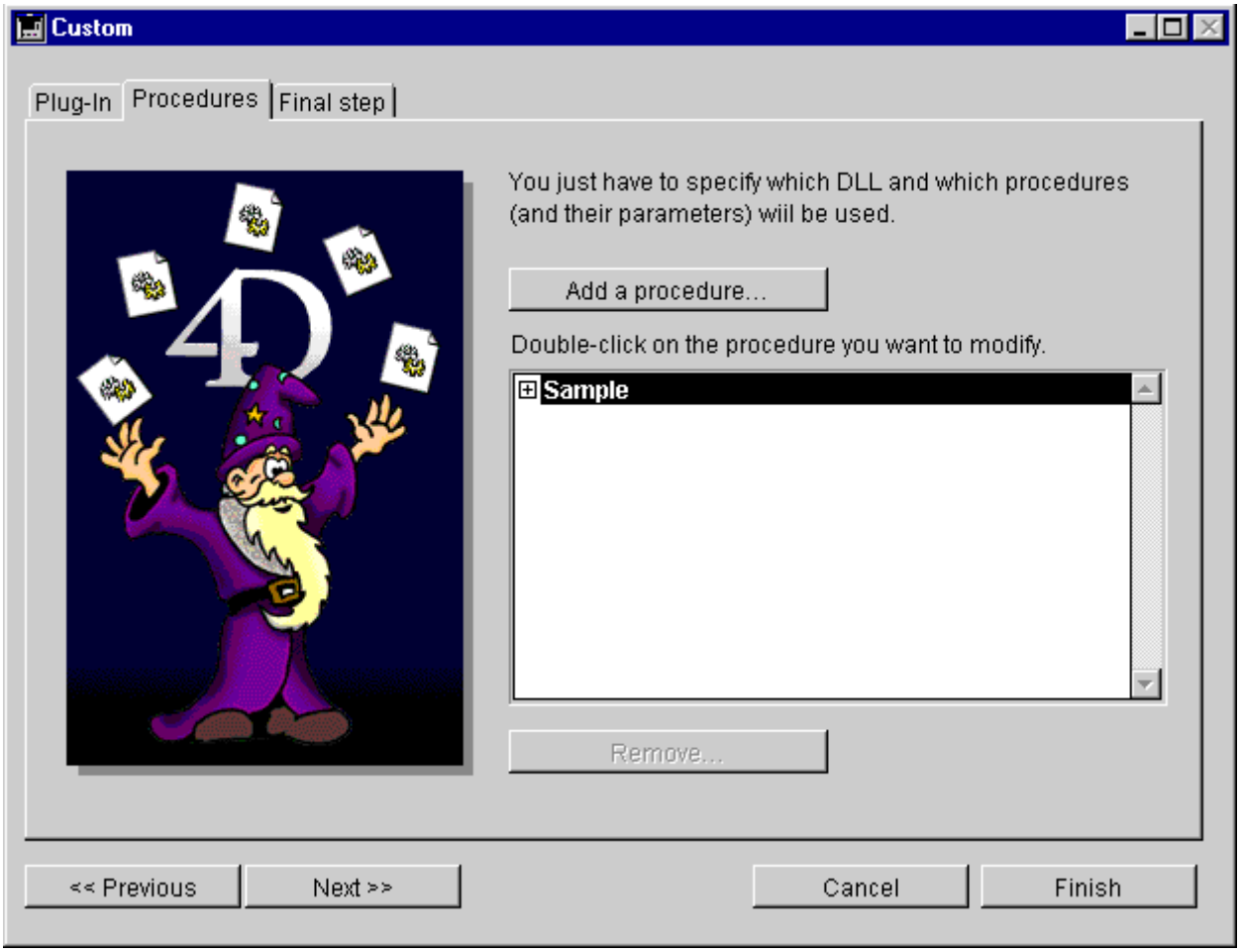
The GetCommandLine function returns a pointer to the command-line string for the current process.

LPTSTR GetCommandLine(VOID)

Select New in the File menu.
In the Plug-in pane, give a name to the plug-in (Sample).



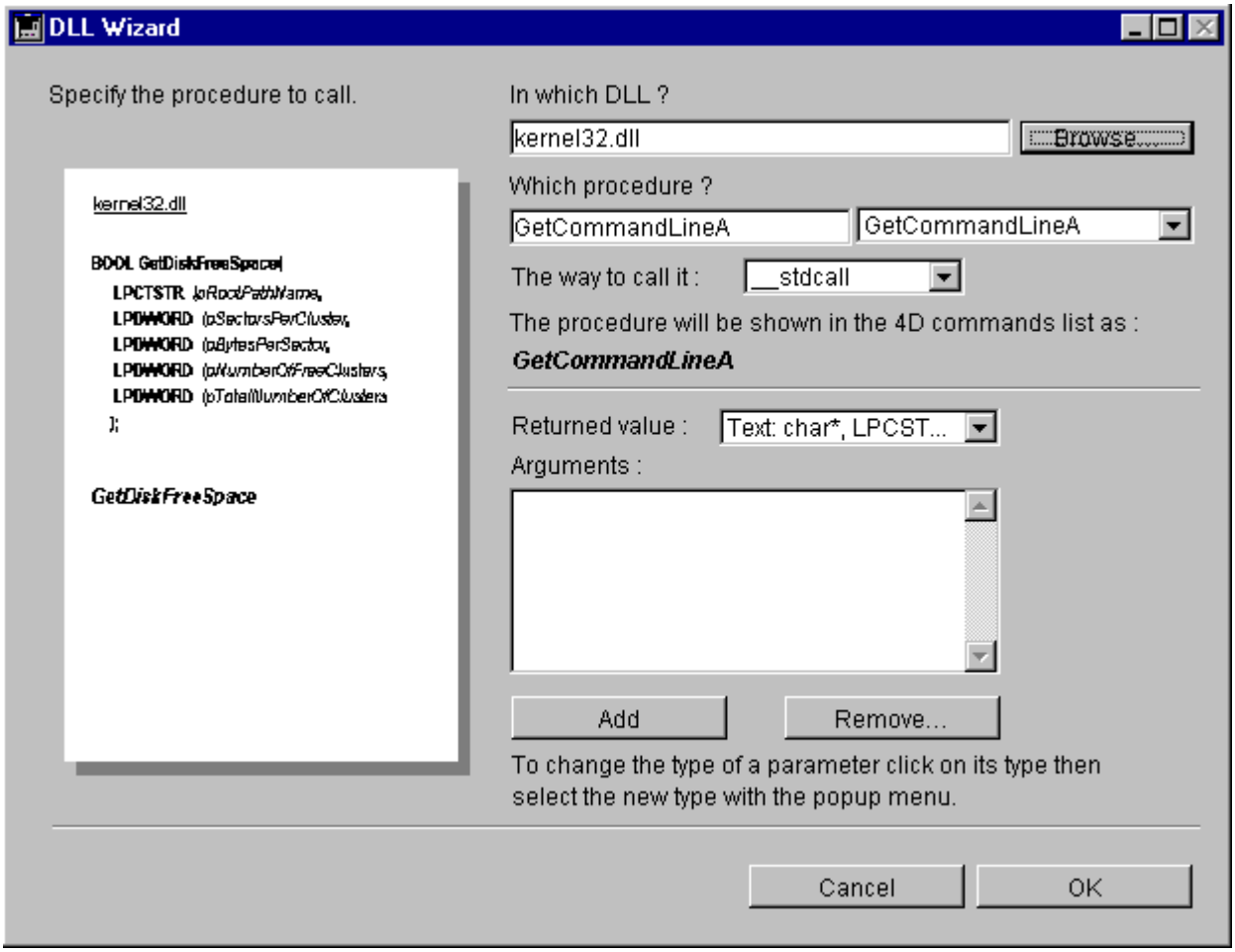
In Procedures window, click on Add routine .



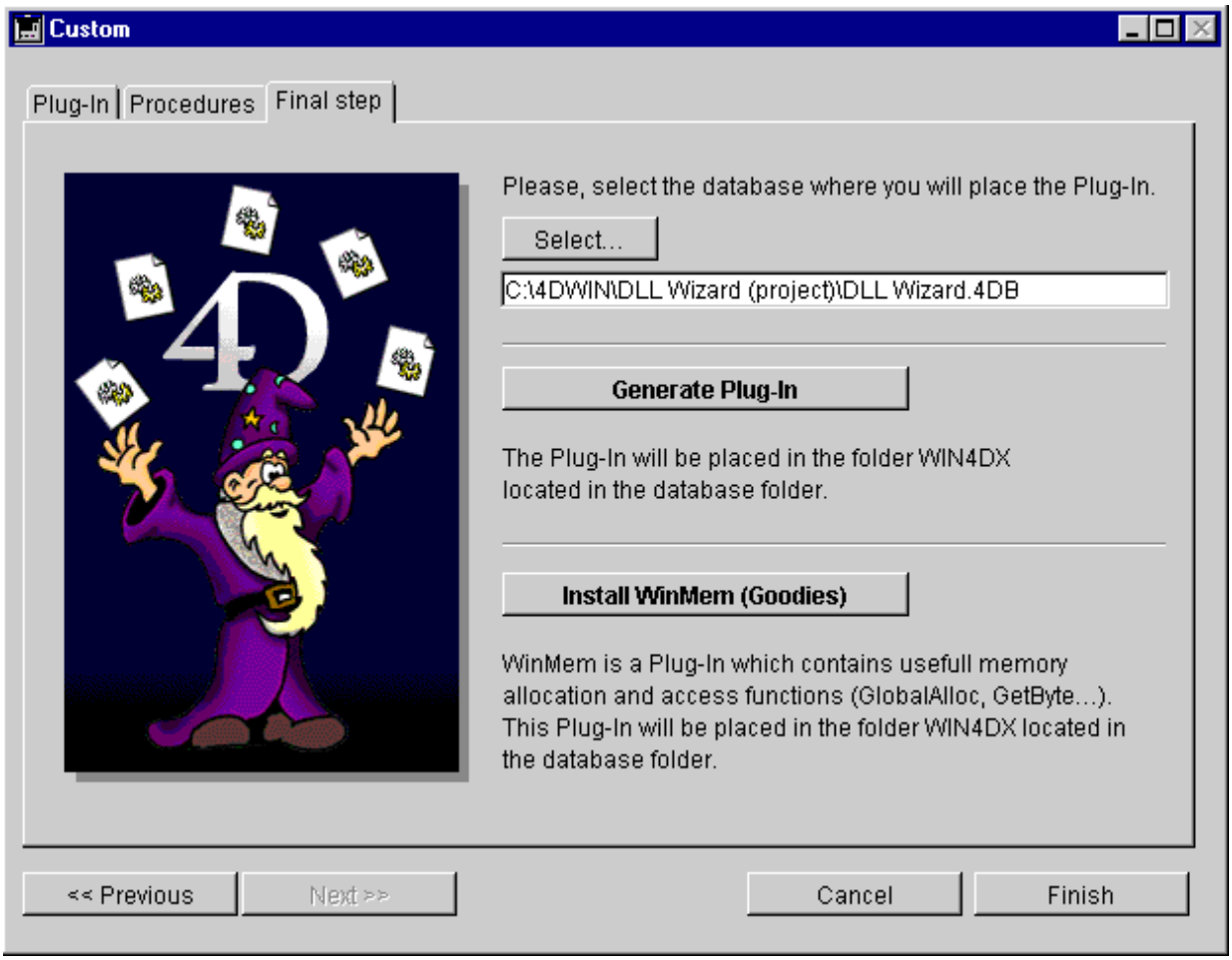
The routine entry form allows you to set the parameters and the returned value of a routine:

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

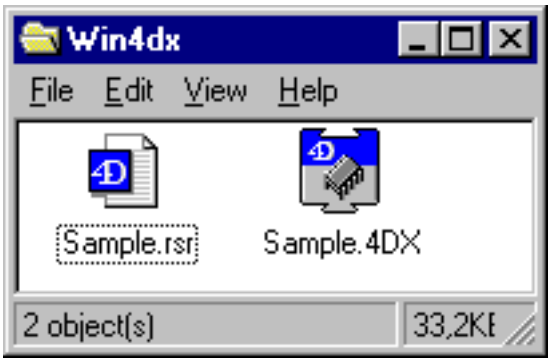
Validate the modifications.



In the Final step window, select the database for which the plug-in is designed.
Then click on Generate Plug-In .



Look in the folder WIN4DX (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
```

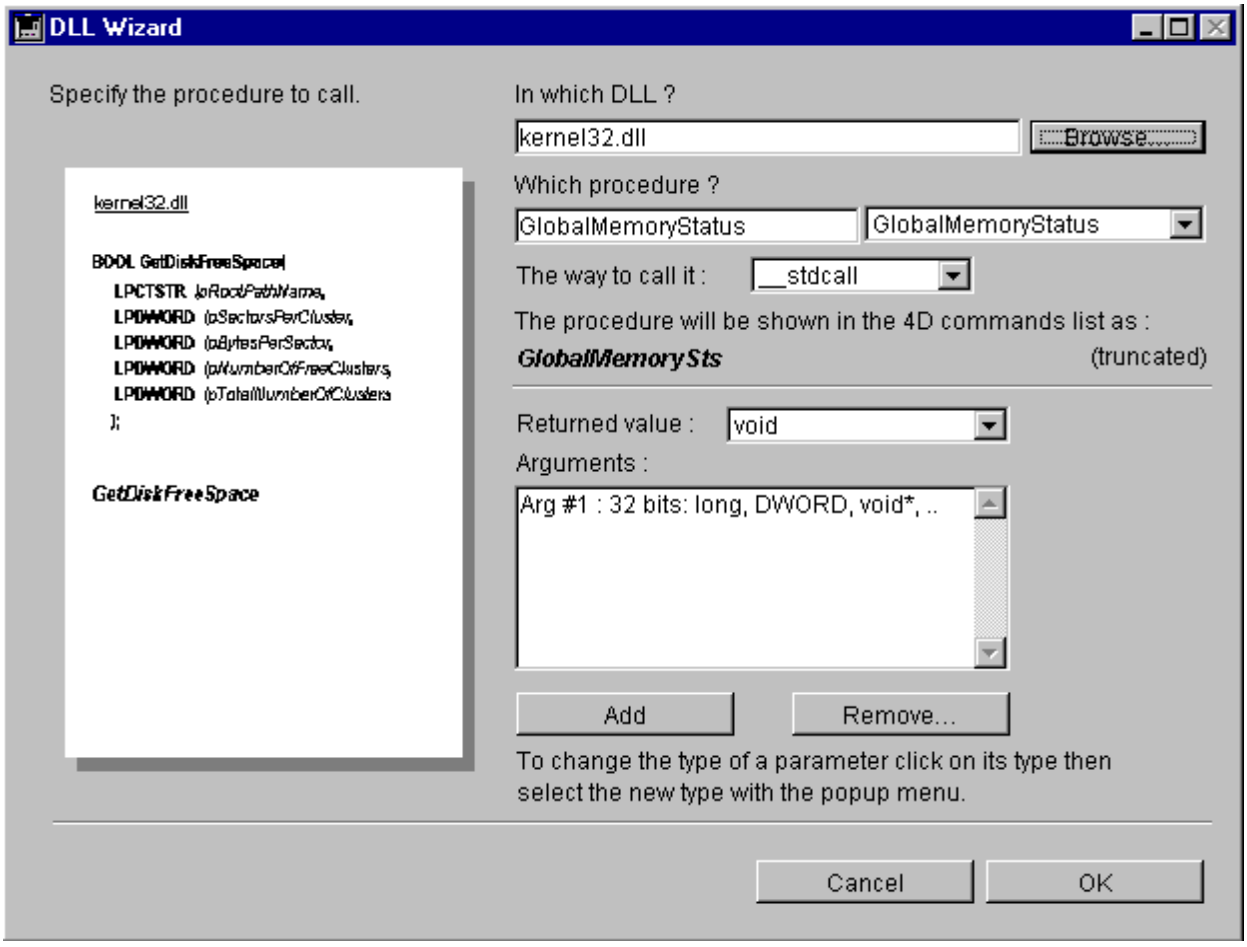
Handling a structure

We plan to use `GlobalMemoryStatus` which 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, *LPMEMORYSTATUS;
```

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



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

In a 4D method the call will be performed as follow:

```
  ` 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
GlobalMemorySus     ($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)
```

DLLs and functions

DLL name
If you specify only the name of the DLL, 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 located, you will get an alert "XXX.dll not loaded". Normally this shouldn't happen with DLLs such as Kernel32.dll, User32.dll,...

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 either in a Windows programming book or in the on-line help of a development system, like Visual or Borland C++.
If you're using a specific DLL (for example a fax software), you can have a look in its 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 have a doubt in which DLL is a procedure, 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 convention (__stdCall).
If your DLL uses C convention, select __cdecl.

Parameters and 4D variables types

You can declare the following parameter types:

8 bits:

```
char, BOOL, BYTE, CHAR, UCHAR, BOOLEAN, CCHAR
```

16 bits:

```
short, WORD, UWORD, SHORT, USHORT
```

32 bits:

```
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, long double
```

float values (32 bits):

```
float
```

void return:

```
void, VOID
```

If your function accepts a parameter type which does not belong to this list, you should pass a parameter compatible type. If your function waits for a pointer to a structure, you need to declare it as a 32 bit pointer, like void*.

Particular case: Structure parameters

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

For example: Calling a function which accepts a RECT parameter.
Original declaration:

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

RECT is declared in 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 handle automatically some conversions for you, like converting a real to a long.

Here are the possible 4D variables type you can pass for each declared type.

8 bits: (`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 bits: (`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 bits: (`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 bigger strings, you can use a text variable. The limitation for text variable is 32000 characters.
Because 4D uses internally Macintosh extended characters, the strings passed to the DLL functions will be automatically converted in ANSI before to call the function, and reconverted into Macintosh Ascii code after.

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 convertions because 4D real variable are double.

double values (64 bits): (`double, long double`)

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).

WinMem plug-in

This plug-in provided by DLL Wizard contains useful functions to allocate global handles or pointers.
These functions are designed to give you an easy access to basic Windows memory management routines.

WinMem plug-in contains also functions that will allow you to read or write data inside the allocated blocks (see WinMem: Access theme).

WinMem: Allocation

WM GlobalAlloc (Flags; Size)->Long

Flags	LongExpr	Object allocation attributes
Size	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 flags parameters (also given as constants in 4D Explorer) specify the attributes of the allocated block. 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 return 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 is discardable. 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 (Handle)->Long

Handle	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 (Handle)->Long

Handle	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's 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 the objects allocated using the GMEM_MOVEABLE flag.

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

_____

WM GlobalUnlock (Handle)->Long

Handle	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 the 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 (Handle)->Long

Handle	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 can't rely on this size to compute by example the number of elements you stored in the block.

WM GlobalReAlloc (Handle; Size; Flags)->Long

Handle	LongExpr	Handle to the global memory object
Size	LongExpr	New size of the block
Flags	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.

Remark: 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, by example, the number of elements you stored in the block.

WinMem: Access

These functions allow you to put specific values anywhere in 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 you deal with a structure, do not forget to respect the byte alignment.

WM SET BYTE (Pointer; Value)

Address	LongExpr	Memory Address
Value	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 in the range 0 to 255.

If you want to pass a signed value, in the range -128 to +127, it's up to you to convert your value before to pass it to the routine, using this conversion algorithm:

```
if ( value<0)
  value:=256+value
end if
WM SET BYTE (pt;value)
```

WM SET SHORT (Pointer; Value)

Address	LongExpr	Memory Address
Value	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 it can hold a value from 0 to 65535.

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

```
if ( value<0)
  value:=65536+value
end if
WM SET SHORT (pt;value)
```

WM SET LONG (Pointer; Value)

Address	LongExpr	Memory Address
Value	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 within -2147483648 to +2147483647.

WM SET DOUBLE (Pointer; Value)

Address	LongExpr	Memory Address
Value	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 range from 1.7 E-308 to 1.7 E+308 with 15 significant digits.

For example:

```
WM SET DOUBLE ( pt; 3.5667 )
```

WM SET FLOAT (Pointer; Value)

Address	LongExpr	Memory Address
Value	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 convertions because 4D real variable are double.

For example:

```
WM SET FLOAT ( pt; 3.5667 )
```

WM SET CSTRING (Pointer; Text)

Address	LongExpr	Memory Address
Text	TextExpr	Text to copy

This command will copy at the specified address the specified text.
This text will be converted into ANSI characters (4D use internally the Mac Ascii code representation). A null character will be added at the end of the text, because most of the DLL functions are waiting Null terminated strings.
If your function is waiting a UNICODE string, you will need first to call a DLL function that handles this conversion, like WideCharToMultiByte, located in Kernel32.DLL.

Example:

```
WM SET CSTRING    ( pt; "Hello World" )
```

WM SET PSTRING (Pointer; Text)

Address	LongExpr	Memory Address
Text	TextExpr	Text to copy

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

Example:

```
WM SetPString    ( pt; "Hello World" )
```

WM SET STRING (Pointer; Text)

Address	LongExpr	Memory Address
Text	TextExpr	Text to copy

This command will copy at the specified address the specified text.
This text will be converted into ANSI characters (4D use internally the Mac Ascii code representation). Neither a leading length byte nor a trailing null character will be added to the text. If your function is waiting a UNICODE string, you will need first to call a DLL function that handles this conversion, like WideCharToMultiByte, located in Kernel32.DLL.

Example:

```
WM SET STRING    ( pt; "Hello World" )
```

WM SET BLOB (Pointer; Value; Length)

Address	LongExpr	Memory Address
Value	BlobExpr	Bytes to copy
Length	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 (Pointer)->Value

Address	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, it is up to you to 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 (Pointer)->Value

Address	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, it is up to you to 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 (Pointer)->Value
Address 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 (Pointer)->Value
Address 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.

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

WM Get float (Pointer)->Value
Address 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 convertions because 4D real variable are double.

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

WM Get cstring (Pointer)->Text
Address LongExpr Memory Address

This command will return a copy of the text stored at the specified address, assuming it's a C string. This text will be converted from ANSI to Macintosh characters (4D use internally the Mac Ascii code representation).
If you want to read a UNICODE string, you need to call first a DLL function to handle the conversion, like MultiByteToWideChar, located in Kernel32.DLL.

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

WM Get pstring (Pointer)->Text

Address	LongExpr	Memory Address

This command will return a copy of the text stored at the specified address, assuming it's a Pascal string. This text will be converted from ANSI to Macintosh characters (4D use internally the Mac Ascii code representation). If you want to read a UNICODE string, you need to call first a DLL function to handle the conversion, like MultiByteToWideChar, located in Kernel32.DLL.

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

WM Get string (Pointer;Length)->Text

Address	LongExpr	Memory Address
Length	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 use internally the Mac Ascii code representation). If you want to read a UNICODE string, you need to call first a DLL function to handle the conversion, like MultiByteToWideChar, located in Kernel32.DLL.

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

WM Get BLOB (Pointer; Length)->Blob

Address	LongExpr	Memory Address
Length	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.

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