

## **DLLs and Memory Management**

Calling Procedures in Dynamic-link Libraries (DLLs)

cd Domain Error and GetLastError information

Memory Management

Calling J.DLL

## Calling Procedures in Dynamic-link Libraries (DLLs)

### Calling procedures incorrectly can crash your system or corrupt memory.

To learn how to call DLLs, run Labs 'DLL: Writing and Using a DLL' and 'DLL: Using System DLLs (file examples)'.

A DLL is a file (usually with extension .dll) that contains procedures. J can call DLL procedures.

Win32 API system services are provided by system DLLs such as *kernel32*. You can also use 3rd party DLLs or DLLs you write yourself.

Script `main\dll.ijs ( load 'dll' )` defines utilities for working with DLLs.

Verb `cd` calls a procedure. The form is:

```
'filename procedure [+][%] declaration' cd parameters
```

- **filename** is the name of the DLL. If a suffix is not provided, .dll is used. The search path for finding a filename that is not fully qualified involves many directories and is different on each platform. Except for system DLLs, a fully qualified filename is recommended.
- **procedure** is the case-sensitive name of the procedure to call. A procedure name that is a number is specified by # followed by digits.

Win32 API procedures that take string parameters are documented under a name, but are implemented under the name with an A suffix for 8 bit characters and a W suffix for 16 bit characters. For example, `CreateFile` is documented, but the procedures you call are `CreateFileA` or `CreateFileW`.

A procedure returns a scalar result and takes 0 or more parameters. Parameters are passed by value or by a pointer to values. Pointer parameters can be read and set.

- **+** option selects the alternate calling convention. The calling convention is the rules for how the result and parameters are passed between the caller and the procedure. Using the wrong one can crash or corrupt memory. J supports two: `__stdcall` and `__cdecl`. `__stdcall` is used by the Win32 API and most procedures. `__cdecl` is the standard C calling convention and is used for some procedures. `__stdcall` is the standard cd calling convention and `__cdecl` is the alternate.
- **%** option does an `fpreset` (floating-point state reset) after the call. Some procedures leave floating-point in an invalid state that causes a crash at some later time. DLL's built with Delphi likely have this problem. If J crashes on simple expressions after calling a procedure, try adding the % option.
- **declaration** is a set of blank delimited codes describing result and parameter types:
  - c character (1 byte)
  - s short integer (2 byte)
  - i integer (4 byte)
  - f short floating-point (4 byte)

d floating point (8 byte)  
 j complex (16 byte) (not as result)  
 \* pointer  
 n no result (result, if any, is ignored and 0 is returned)

The first declaration type describes the result and the remaining ones describe the parameters in the `cd` right argument.

The `c i d` and `j` types are native J types and the `s` and `f` types are not. Scalar `s` and `f` values are handled as `i` and `d` types. Pointer `s` and `f` parameters are handled as character data.

The `*` type is a pointer to values. A `*` can be followed by `c s i f d` or `j` to indicate the type of values. The DLL can read from this memory or write to it.

A scalar type (`c s i f d j`) must have a scalar parameter. A pointer type (`* *c *s *i *f *d *j`) must have either a non-scalar parameter of the right type, or a boxed scalar integer that is a memory address.

J boolean data is stored as 1 byte values. Boolean parameters are automatically converted to integers.

The `mema` result (Memory Management) can be used as a `*` type parameter. A memory address parameter is a boxed scalar. The NULL pointer is `<0`.

The `cd` right argument is a list of enclosed parameters. An empty vector is treated as 0 parameters and a scalar is treated as a single parameter.

The `cd` result is the procedure result catenated with its possibly modified right argument.

For example, the Win32 API procedure `GetProfileString` in `kernel32` gets the value of the `windows/device` keyword.

```
a=: 'kernel32 GetProfileStringA s *c *c *c *c s'
b=: 'windows';'device'; 'default'; (32$'z');32
a cd b
+---+-----+-----+-----+-----+-----+-----+-----+-----+
|31|windows|device|default|HP LaserJet 4P/4MP, HPPCL5MS, LPT |32|
+---+-----+-----+-----+-----+-----+-----+-----+-----+
```

The first type `s` indicates that the procedure returns a short integer. The first pointer names a section. The second pointer names a keyword. The third pointer is the default if the keyword is not found. The fourth parameter is where the keyword text is put. The fifth parameter is the length of the fourth parameter.

If the `GetProfileStringA` declaration was wrong, say a `d` result instead of `s`, it would crash your system. If the fifth parameter was 64 and the keyword was longer than the 32 characters allocated by the fourth parameter, the extra data would overwrite memory.

Procedures are usually documented with a C prototype or a Visual Basic declaration. The C prototype and VB declaration for `GetProfileString` are:

```
DWORD GetProfileString(
```



## **cd Domain Error and GetLastError Information**

`cderr` ' ' returns information about a cd domain error:

0 0 - no error

1 0 - file not found

2 0 - procedure not found

3 0 - too many DLLs loaded (max 20)

4 0 - parameter count doesn't match declarations

5 x - declaration x invalid

6 x - parameter x type doesn't match declaration

`cderrx` ' ' returns `GetLastError` and text from the last cd.

## Memory Management

Some DLL procedures return pointers to memory or require parameters of pointers to memory that cannot be provided by referencing a J array. The following verbs, defined in main\dll.ijs, are provided to allocate, free, read and write memory:

```
mema  allocate bytes of memory
memr  read bytes from memory (as type)
memw  write bytes to memory (from type)
memf  free memory allocated by mema
```

`mema` allocates memory. The result is an integer memory address. A 0 result indicates the allocation failed. For example:

```
address=: mema length
```

`memf` frees memory. The argument could be a `mema` result or pointer returned by a procedure. A 0 result is success and a 1 is failure.

```
memf address
```

`memr` reads data from memory. A `_1` count reads up to the first 0 (read a C string).

```
data=: memr address,byte_offset,count [,type]
```

`memw` writes data to memory. If `type` is `char`, `count` can be 1 greater than the length of the string left argument, in which case a 0 is appended (writing a C string).

```
data memw address,byte_offset,count [,type]
```

`memr` and `memw` `type` parameter is 2 4 8 or 16 for char integer float or complex. The default is 2. The `count` parameter is a count of elements of the type.

## Calling J.DLL

The J DLL can be called by any program that can call DLLs.

Since J.DLL is itself used by the J session, you need to make a copy of J.DLL first before calling it from J; for example, copy it to file JJ.DLL.

File system\examples\data\jdll.h. gives C prototypes for J procedures.

Script system\examples\dll\jdll.ijs. gives examples of calling the J DLL from J.

Use procedure JDo to execute a sentence. For example, the following writes text abc to file t1.txt:

```
load 'dll files'
cmd=: ''abc'' 1!:2 <'t1.txt'' NB. example sentence
'jj.dll JDo i *c' cd <cmd      NB. send to J DLL
+-----+
|0|'abc' 1!:2 <'t1.txt'|
+-----+
fread 't1.txt'                NB. check file was written
abc
```

Use procedure JGetM to retrieve a J variable. The cd right argument is a name, followed by 4 pointers, which will correspond to the result datatype, rank, pointer to shape, and pointer to values. For example:

```
'jj.dll JDo i *c' cd <'ABC=: i.5' NB. define ABC
+-----+
|0|ABC=: i.5|
+-----+

'jj.dll JGetM i *c *i *i *i *i' cd 'ABC';4#<,0
+-----+-----+
|0|ABC|4|1|13496196|13496200|
+-----+-----+
```

The 6 items in the result are: error code (0=success), name, datatype (4=integer), rank(1), pointer to shape, and pointer to values.

The pointers refer to memory addresses within the J DLL. You should reference their values before calling the J DLL again, as further calls may invalidate the memory addresses. Use function memr to read memory and and ic to convert to J integers. For example, the shape is:

```
_2 ic memr 13496196 0 4
5
```

Once the result datatype and shape are known, you can read the values, again using memr, and convert to a J variable.

File system\examples\dll\jdll.ijs defines functions that perform the necessary conversions. For example:

```
load 'system\examples\dll\jdll.ijs'
```

```
      jdo 'ABC=: i.3 4'  
+-----+  
|0|ABC=: i.3 4|  
+-----+
```

```
      jget 'ABC'  
0 1 2 3  
4 5 6 7  
8 9 10 11
```

```
      jcmd 'q: 123456'  
2 2 2 2 2 2 3 643
```