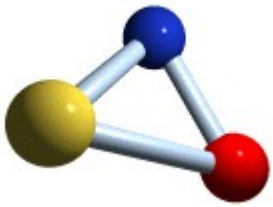
Numeric Expression Engine v3.3



T. Radde

DISCLAIMER:

IDEAL Software and T. Radde hereby disclaim any warranty, either expressed or implied, as to the suitability of this product for any purpose whatsoever. IDEAL Software and T. Radde will not be responsible for any damage, loss of data, incorrect data, or other problems arising from the use or misuse of this software.
By using this product, the user agrees to accept responsibility for any problems, and to hold IDEAL Software and T. Radde free of any liability (NOTE: You should include a disclaimer of your own with any files you distribute).

THE INFORMATION AND CODE PROVIDED HEREUNDER (COLLECTIVELY REFERRED TO AS "SOFTWARE") IS PROVIDED "AS IS," WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL IDEAL SOFTWARE AND/OR T. RADDE BE LIABLE FOR ANY DAMAGES WHATSOEVER INCLUDING DIRECT, INDIRECT, INCIDENTAL, CONSEQUENTIAL, LOSS OF BUSINESS PROFITS, OR SPECIAL DAMAGES AS THE RESULT OF USING THIS PROGRAM, EVEN IF IDEAL SOFTWARE AND/OR T. RADDE HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

USING (RUNNING) THIS SOFTWARE ACKNOWLEDGES YOUR ACCEPTANCE OF THIS AGREEMENT.

License Agreement:

You agree to include a copyright notice like <"Expression Engine" Copyright © by IDEAL Software, T. Radde> in your "about dialog" or the help-file of your software (if they exist) or in your documentation.

You agree not to release documentation concerning how to control VPE from any programming language (i.e. description of the function call interface or any flags) to third parties.

When using the full licensed version:

The only redistributable files are: EXPENGIN.DLL (EXPENG32.DLL) (EXPENOS2.DLL)
ExEngine may only be given away to third parties as an integrated part of your software. must have significant main functions other than ExEngine.

This license gives you the possibility to integrate ExEngine in your products and give it to third parties. But these third parties have no right to give ExEngine away to other third parties. If you have such needs, contact T. Radde for special conditions.

USING (RUNNING) THIS SOFTWARE ACKNOWLEDGES YOUR ACCEPTANCE OF THIS AGREEMENT.

Notes:

This demonstration program is freeware. Give it to your friends, colleagues and anyone else who may be interested.

You must distribute this program unchanged and intact; all files must be included with it, including "expengin.wri".

Freeware/Shareware vendors may distribute this version of the program freely.

ExpEngine is available for Win 16 bit, Win 32 bit (Win95 and NT) and OS/2.

ExpEngine is available as a trial and development version for $99 per platform. Please note: the $99 version is NOT license-free. The trial version grants you the right to use ExpEngine ONLY in-house in one location. If you want to distribute ExpEngine within your products, or if you want to use it in different office branches, you will need to license it. The price for the licensed version is $499 per platform. This allows you to make as many copies as you like (embedded within your products - see "License Agreement"). If you purchase the $99 version first, you may upgrade to the full license later for $400 (current).

Order by check or money order. Checks from outside Germany: Please send Eurocheck in DM currency. Otherwise be so kind as to add $15 to the sum; that is, the bank's foreign money transactions charge. If you can't receive ExpEngine via CIS e-mail (currently I have no Internet binary support), or if you wish to receive ExpEngine for any other circumstances via mail, please add $10 for packaging and shipping (see pricing on next page).

Pricing per Platform:

	Unlicensed	**Licensed**
Base Price	$99	$499
Outside Germany, no Eurocheck in DM	+ $15	+ $15
Via CIS e-mail	free	free
Via mail	+ $10	+ $10

All prices as of 2/96

Win95 and Win-NT = ONE platform (Win32).
Both versions (unlicensed and licensed) come with the source code of express.exe for all three platforms.

I'm also looking for distributors in all countries worldwide.

My address:
T. Radde
Grefrather Weg 96
41464 Neuss
Germany

CompuServe: 100 430, 34 27

Internet: 100430.3427@compuserve.com

Files in this archive:

analysis.exp	- a demo configuration file for express.exe
default.exp	- an empty demo configuration file for express.exe
expengin.wri	- this file
express.exe	- demo program win 16
express.ini	- ini-file for express.exe for all three platforms
trigonom.exp	- a demo configuration file for express.exe

Introduction

ExEngine is a DLL, which can be used from any programming language that supports the DLL-calling conventions. (like C/C++, Pascal, Basic, SQLWindows and so on). It gives you the ability to define and modify variables and to use them within your self-defined functions at runtime. Using it from compiler-languages, such as C, you are additionally able to EXPORT THE ADDRESSES OF VARIABLES AND FUNCTIONS (!!!) FROM WITHIN YOUR CODE TO THE DLL, so that ExEngine can directly access them. This provides you nearly unlimited possibilities.

Since ExEngine generates pseudo-code, computations are done very fast. (All computations are done with the "double"-datatype -this is not the best for fast drawing graphs, but for calculations.)
Note, that all graphs drawn by Express.Exe are computed during they are painted. There are no arrays or other tricks to increase the performance.

The number of variables (exported or created) and the number of stored expressions is only limited by available memory.

Examples on using ExEngine from C / C++:

```
void func()
{
  long   pcode_handle;
  double x, result;

  ExpBindVar(hExp, "x", &x);      // rts the variable's address to engine
  if (ExpGetErr(hExp))
        handle_error();           // perhaps the variable "x" is already defined ?
                                  // (lack of memory problems are handled also)

  // just compiles - the storage of the pcode is managed by the engine
  pcode_handle = ExpParseAndStore(hExp, "sin(x * 3) + sin(x)");
  if (ExpGetErr(hExp))
        handle_error();           // try possible error-conditions with the "Expression Dialog"
                                  // all error messages are directly generated by the engine!
                                  // (lack of memory problems are handled also)

  for (x = 0; x <= 1; x += 0.01)
  {
    result = ExpEval(hExp, pcode_handle);     // execute the pcode - all current variable values are used
    if (ExpGetErr(hExp))
          handle_error();         // perhaps division by zero or something else...
    printf("f(%g) = %g\n", x, result);
  }

  ExpRemoveVar(hExp, "x");
}
```

```
void func2()
{
  double result;

  ExpDefineVar(hExp, "x", 3.14);  // creates variable within the engine, initial value is 3.14
  if (ExpGetErr(hExp))
        handle_error();           // perhaps the variable "x" is already defined?
                                  // (lack of memory problems are handled also)

  result = ExpParseAndEval(hExp, "sin(x * 3) + sin(x)");    // directly compiles and evaluates
  if (ExpGetErr(hExp))
        handle_error();           // perhaps the variable "x" is already defined ?
                                  // (lack of memory problems are handled also)

  printf("for x=%lg the result is %lg", ExpGetVar(hExp, "x"), result);

  ExpSetVar(hExp, "x", 1.57);               // sets variable x to value 1.57
  result = ExpParseAndEval(hExp, "x^3 + 5.43");
  if (ExpGetErr(hExp))
        handle_error();           // perhaps division by zero or something else...
  printf("for x=%lg the result is %lg", ExpGetVar(hExp, "x"), result);
  ExpRemoveVar(hExp, "x");                  // remove variable definition from engine
}
```

Windows 16- and 32-bit:

Before including EXPENGIN.H you need to define the macro WINDOWS_DLL as below.

```
#define WINDOWS_DLL     // if any Windows-DLL (16 / 32 bit)
```

OS/2:

Before including EXPENGIN.H you need to define the macro OS2_DLL as below.

```
#define OS2_DLL         // if an OS2-DLL
```

Explanation of the demo ”express.exe”:

The program is divided into two main parts:
1) Graph drawing (Menu: "Actions / Draw Graph")
2) Computing Expressions (Menu: "Actions / Expressions")

In both dialogs you can define expressions and other parameters.
All these definitions can be saved using the Menu: "File / Save" and loaded using the Menu: "File / Open".

Note: The "save"-function also saves the current layout - i.e. the positions and sizes of the windows.
But the last saved layout is always used for ALL definition files.
On startup Expression automatically loads the last saved definition file.

The rest of the program is explaining itself.

-Just keep in mind, that changes only become valid, after you pushed the "Insert"- or "Change"- or "Set"-button...

-Note the "y-range" button in the "Draw Graph" dialog, which calculates for all drawn functions over the specified x-range the maximum y-range, to make all graphs fit into the coordinate system.

-Also note, that you can enter real EXPRESSIONS in both, the x-unit and the y-unit fields - you have two predefined variables:
"xr" and "yr", which means the value of "x-range" and the value of"y-range".
This gives you the following advantage:
Imagine, you want to draw the sine-function over PI (=3.1415...). So you enter 3.1415 in the x-range-field. If you want to have the x-units drawn on PI/4 you would need to calculate these values.
Why? - We have got an expression-evaluator build in!
So just enter in the x-unit-field: "xr / 4"
That's all. - Look at the "trigonom.exp" - example definition.

How to use ExpEngine:

1) Initialise the engine with ExpInit() (retrieve a handle to your local definitions space)

2) Export addresses of variables from your source to the engine with ExpBindVar()

3) Create a variable within the engine (interpreter like) with ExpDefineVar()

4) Export addresses of functions from your source to the engine with ExpBindFunc()

5) Compile and store the p(seudo)-code with ExpParseAndStore()

6) Evaluate the p-code (with all actual values of all bound and / or defined variables) with ExpEval()

7) Check for possible errors always with ExpGetErr()

8) Retrieve clear error text for the user with ExpGetErrText() [english] or ExpGetErrTextGerman() [german]

9) Close the local definitions space with ExpExit()

Functions (in functional order):

Remark: NUMTYPE is defined as “double” datatype.

The reason for this definition is, that we're also able to generate engines with other numeric types for special circumstances.
This will not apply to this standard edition of ExEngine.

int ExpGetVersion();

Action: return the current version number coded as follows

Parameters: none

Returns: the current version number coded as follows
Hi-Byte = major no. (0-99) / Lo-Byte = minor no. (0-99)

for example: 0x0115 = 1.21

long ExpInit();

Action: Initialize the engine. Retrieve a handle to your local definitions space.
This handle has to supplied to all other function calls

Parameters: -

Returns: handle to your local definitions space **(hExp)**

void ExpExit(long hExp);

Action: free your local definitions space.
It is strongly recommended to call this function before terminating your application.
Otherwise used memory might not be freed to the operating system.

Parameters: handle to your local definitions space

Returns: -

int ExpGetErr(long hExp);

Action: retrieve the errorcode generated by the last Exp-function call

Parameters: handle to your local definitions space

Returns: errorcode (0=ok, else see chapter "Error Codes")

int ExpIsEvalErr(long hExp);

Action: check wether the error was caused by an evaluation-function-call

Parameters: handle to your local definitions space

Returns: TRUE if so, else FALSE

int ExpGetErrPos(long hExp);

Action: get the character position within the string to parse where the error occurred
this function only returns valid data, if the last error was caused by a parse-function

Parameters: handle to your local definitions space

Returns: character position

char * ExpGetErrText(long hExp);

Action: retrieve the clear error text for the user during runtime in english language

Parameters: handle to your local definitions space

Returns: pointer to text

char * ExpGetErrTextGerman(long hExp);

Action: retrieve the clear error text for the user during runtime in german language

Parameters: handle to your local definitions space

Returns: pointer to text

int ExpBindVar(long hExp, char *name, NUMTYPE *address);

Action: export the address of a variable from your source to the engine
this can normally only be done with compiler languages

Parameters: handle to your local definitions space, the name the user can use within the expressions, the address of the variable

Returns: 0 = Ok, < 0 = Errorcode

long ExpDefineVar(long hExp, char *name, NUMTYPE init_value);

Action: Create a variable within the engine (interpreter like)

Parameters: handle to your local definitions space, the name the user can use within the expressions, initialization value for the variable

Returns: >=0 a handle (ID) to the variable (for much faster access when setting / retrieving values), < 0 = Errorcode

int ExpSetVar(long hExp, char *name, NUMTYPE value);

Action: Set the value of a defined variable within the engine (interpreter like)

Parameters: handle to your local definitions space, the name the user can use within the expressions, new value for the variable

Returns: 0 = Ok, < 0 = Errorcode

void ExpIdSetVar(long hExp, long id, NUMTYPE value);

Action: Set the value of a defined variable within the engine (interpreter like) , by ID

Parameters: handle to your local definitions space, ID of the variable (previously returned by ExpDefineVar()), new value for the variable

Returns: -
Here no error can occur, ExpErr is always set to zero, if ID is invalid, your application will crash

NUMTYPE ExpGetVar(long hExp, char *name);

Action: Retrieve the value of a defined variable within the engine (interpreter like)

Parameters: handle to your local definitions space, the name the user can use within the expressions

Returns: The value of the variable or 0 and ExpErr is set (to VARERR)

NUMTYPE ExpIdGetVar(long hExp, long id);

Action: Retrieve the value of a defined variable within the engine (interpreter like)

Parameters: handle to your local definitions space, the name the user can use within the expressions

Returns: -
Here no error can occur, ExpErr is always set to zero, if ID is invalid, your application will crash

int   ExpRemoveVar(long hExp, char *name);

Action: Remove defined **AND** bound variables from the engine (free memory)
Each bound variable needs memory within the engine. If you remove a bound variable, this memory is freed. Of course the variable itself is untouched.

Parameters: handle to your local definitions space, the name the user can use within the expressions

Returns: 0 = Ok, < 0 = Errorcode

void ExpIdRemoveVar(long hExp, long id);

Action: Remove a defined variable from the engine (free memory), by ID

Parameters: handle to your local definitions space, ID of the variable (previously returned by ExpDefineVar())

Returns: -
Here no error can occur, ExpErr is always set to zero, if ID is invalid, your application will crash

void ExpStartQueryFunc(long hExp);

Action: Prepares the engine to enumerate all known functions (bound and internal) to your application via ExpGetFunc()
See chapter "Build-In Functions"

Parameters: handle to your local definitions space

Returns: -

int   ExpGetFunc(long hExp, char *buf, int bufsize);

Action: Retrieve with each call the next known function (bound and internal) from the engine
You shouldn't call any other engine-function until the enumeration is finished!
See chapter "Build-In Functions"

Parameters: handle to your local definitions space, pointer to a buffer to retrieve the function name, maximum size of the buffer

Returns: TRUE = Ok, FALSE = no more functions

int   ExpBindFunc(long hExp, char *name, PTRFN address);

Action: export the address of a function from your source to the engine
This can normally only be done with compiler languages.
Bind-functions always have to receive exactly one parameter of type NUMTYPE.

Parameters: handle to your local definitions space, the name the user can use within the expressions, the address of the function

Returns: 0 = Ok, < 0 = Errorcode

int   ExpRemoveFunc(long hExp, char *name);

Action: Remove bound functions from the engine (free memory)

Parameters: handle to your local definitions space, the name the user can use within the expressions

Returns: 0 = Ok, < 0 = Errorcode

long ExpParseAndStore(long hExp, char *expr);

Action: Parses the expression in expr and generates p-code. The p-code is stored in an internal buffer and an ID to this buffer is returned.

Parameters: handle to your local definitions space, string with the expression

Returns: >= 0 ID, < 0 = Errorcode

void ExpReleaseStored(long hExp, long id);

Action: Remove stored p-code from the engine (free memory)

Parameters: handle to your local definitions space, the ID returned by ExpParseAndStore()

Returns: -
Here no error can occur, ExpErr is always set to zero, if ID is invalid, your application will crash

int ExpParse(long hExp, char *expr, char *pcb, int pcbsize);

Action: Parses the expression in expr and generates p-code. The p-code is stored in the buffer pcb that has a maximum size of pcbsize bytes

Parameters: handle to your local definitions space, string with the expression, pointer to a buffer for the generated p-code the caller has to supply, the size of the buffer in bytes

Returns: >= 0 size in bytes of the generated p-code, < 0 = Errorcode

NUMTYPE ExpEval(long hExp, char *pcodebuffer);

Action: Evaluates the p-code in pcodebuffer (with all actual values of all bound and / or defined variables).

Parameters: handle to your local definitions space, pointer or ID to the p-code

Returns: The result of the evaluation, always check for possible error conditions with ExpGetErr()

NUMTYPE ExpParseAndEval(long hExp, char *expr);

Action: Parses the expression in expr and generates p-code in an internal temporary buffer. The p-code is directly executed and then released from memory. This function is only useful, if you have to evaluate an expression once. Otherwise you waste cpu-time and performance of your application by much.

Parameters: handle to your local definitions space, string with the expression

Returns: The result of the evaluation, always check for possible error conditions with ExpGetErr()

Error Codes:

The following constants are defined in expengin.h

See functions ExpGetErrText() [english] or ExpGetErrTextGerman() [german] to retrieve the clear error text for the user during runtime.

Errorcodes generated by PARSE():

CONSTERR	constant value (a number) syntax error
VARERR	variable unknown / mis-spelled
FUNCERR	function unknown / mis-spelled
SYNTERR	common syntax error
BRACEERR	unbalanced braces
PCBERR	PCODE-buffer overflow (the reserved space of the p-code buffer is not big enough)
STKERR	stack-overflow (this is the internal stack of the p-code compiler, possibly too much brace-levels)
ARGERR	missing operand

Errorcodes generated by EVALUATION():

ESTKERR	stack-overflow (this is the internal stack of the p-code machine, possibly too much brace-levels)
DIV0ERR	Division by Zero Error
DOMAINERR	Math.Lib.: Argument out of range (e.g. sqrt(-1))
RANGEERR	Math.Lib.: Result out of range (result to large / small for double representation)
UNKNOWNERR	common MathLib-Exception

Common Errorcodes:

VARNAMEERR	illegal variable name (syntax error)
STORAGEERR	couldn't store variable/function, not enough memory
VARISDEFINEDERR	variable already defined
FUNCNAMEERR	illegal function name (syntax error)
FUNCISDEFINEDERR	function already defined

Build-In Functions and Operators:

Currently ExEngine knows the following operators:

+	plus
-	minus
*	multiply
/	divide
^	power

Currently ExEngine has the following functions build-in:

Name	**C-name**
"abs",	fabs,
"sqrt",	sqrt,
"exp",	exp,
"log",	log,
"log10",	log10,
"cos",	cos,
"sin",	sin,
"tan",	tan,
"acos",	acos,
"asin",	asin,
"atan",	atan,
"cosh",	cosh,
"sinh",	sinh,
"tanh",	tanh,
"ceil",	ceil,
"floor",	floor

Trademarks: