1. About the KCompiler

1.1. Welcome to the KCompiler

KCompiler is an unvisual Delphi component, that lets you extend your application using arithmetical expressions (formulas) such as “4*x/2 - y*z + cos(z) + 5 - Power(z-x, 2) - 3”. This is not a parser, but a compiler, that generates real machine code, optimized for different models and generations of FPUs. Also it can perform deep analyzing of formula and simplify it using mathematical rules (for example, given expression will be simplified to “2 + 2*x - y*z + cos(z) - Power(z-x, 2)” and all operations may be performed parallel due to your settings). Also KCompiler can handle syntax errors and report your application about them. Different options let you generate code optimal for all existing AMD and Intel processors produced since 1995. You also may easily extend KCompiler with your functions (5 types) and variables (10 types). Use KCompiler to calculate formulas that are unknown at the stage of program compilation or to achieve greater performance with your Delphi application.

KCompiler can work now with Delphi 5, 6, 7.

The author can be contacted via E-Mail:
zgonnik@math.dvgu.ru

1.2. How to use this document

This manual explains how to use the KCompiler. It provides information on how to get started with the KCompiler, how this compiler operates, and what capabilities it offers for high performance. You learn how to use the standard and advanced compiler features to gain maximum performance of your application. This documentation assumes that you are familiar with the Delphi programming language and with the basic programming methods like objects and subroutines. It’s also good for you to be familiar with IA-32 assembler but not necessary.

1.3. Features

The features of KCompiler are:

- supported by Delphi 5, 6, 7 compilers
- common arithmetic operations +, -, *, / and parenthesis (,)
- expression simplifying
- deep analyzing of formula
- great speed of calculations
- large set of optimizations
- syntax errors handling
- data alignment
- extendable variable list

- large set of supported variables’ types
- extendable function list
- quick function parameters passing
- smart function call mode
- multithread computations support

1.4. Installation

1) Unzip KCompiler.zip to any suitable directory
2) Select “Component->Install packages…”
3) Choose “Add…”, select file “YOUR_DIRECTORY\Delphi X\KCompiler_DX0.bpl”
4) Enjoy using KCompiler – it will appear in the tab “DOMIN” on your component palette

1.5. License Agreement

KCOMPILER - PRODUCT LICENSE INFORMATION

NOTICE TO USERS: CAREFULLY READ THE FOLLOWING LEGAL AGREEMENT. USE OF THE SOFTWARE PROVIDED WITH THIS AGREEMENT (THE "SOFTWARE") CONSTITUTES YOUR ACCEPTANCE OF THESE TERMS. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT INSTALL AND/OR USE THIS SOFTWARE. USER'S USE OF THIS SOFTWARE IS CONDITIONED UPON COMPLIANCE BY USER WITH THE TERMS OF THIS AGREEMENT.

1. LICENSE GRANT. DOMIN Software grants you a license to use one copy of the version of this SOFTWARE on any one system for as many licenses as you purchase. "You" means the company, entity or individual whose funds are used to pay the license fee. "Use" means storing, loading, installing, executing or displaying the SOFTWARE. You may not modify the SOFTWARE or disable any licensing or control features of the SOFTWARE except as an intended part of the SOFTWARE's programming features. This license is not transferable to any other system, or to another organization or individual. You are expected to use the SOFTWARE on your system and to thoroughly evaluate its usefulness and functionality before making a purchase. This "try before you buy" approach is the ultimate guarantee that the SOFTWARE will perform to your satisfaction; therefore, you understand and agree that there is no refund policy for any purchase of the SOFTWARE.

2. OWNERSHIP. The SOFTWARE is owned and copyrighted by DOMIN Software. Your license confers no title or ownership in the SOFTWARE and should not be construed as a sale of any right in the SOFTWARE.

3. COPYRIGHT. The SOFTWARE is protected by United States copyright law and international treaty provisions. You acknowledge that no title to the intellectual property in the SOFTWARE is transferred to you. You further acknowledge that title and full ownership rights to the SOFTWARE will remain the exclusive property of DOMIN Software and you will not acquire any rights to the SOFTWARE except as expressly set forth in this license. You agree that any copies of the SOFTWARE will contain the same proprietary notices which appear on and in the SOFTWARE.

4. REVERSE ENGINEERING. You agree that you will not attempt to reverse compile, modify, translate, or disassemble the SOFTWARE in whole or in part.

5. NO OTHER WARRANTIES. DOMIN Software DOES NOT WARRANT THAT THE SOFTWARE IS ERROR FREE. DOMIN Software DISCLAIMS ALL OTHER WARRANTIES WITH RESPECT TO THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES OR LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY MAY LAST, OR THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM JURISDICTION TO JURISDICTION.

6. SEVERABILITY. In the event of invalidity of any provision of this license, the parties agree that such invalidity shall not affect the validity of the remaining portions of this license.

7. NO LIABILITY FOR CONSEQUENTIAL DAMAGES. IN NO EVENT SHALL DOMIN Software OR ITS SUPPLIERS BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, SPECIAL, INCIDENTAL OR INDIRECT DAMAGES OF ANY KIND ARISING OUT OF THE DELIVERY, PERFORMANCE OR USE OF THE SOFTWARE, EVEN IF DOMIN Software HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT WILL DOMIN Software LIABILITY FOR ANY CLAIM, WHETHER IN CONTRACT, TORT OR ANY OTHER THEORY OF LIABILITY, EXCEED THE LICENSE FEE PAID BY YOU, IF ANY.

8. ENTIRE AGREEMENT. This is the entire agreement between you and DOMIN Software which supersedes any prior agreement or understanding, whether written or oral, relating to the subject matter of this license.

1.6. Purchasing

KCompiler price is US $49.95 (only compiled component without source).

Please note that refunds are not possible: you have the opportunity to try KCompiler's demo before buy it.

Select one of the following registration ways.

1.6.1. Online registration on the Internet

This is the fastest and easiest way. The ordering page is on a secure (SSL) server, ensuring that your confidential information remains confidential. To purchase KCompiler, you can enter the registration online on the Internet here.

Alternatively, you can go to http://www.shareit.com and enter the program number there: 176176.

1.6.2. Phone, fax, postal mail

If you do not have access to the Internet, you can register via phone, fax or postal mail. Please fill and print out the "**order.frm**" file and fax or mail it to:

element 5 AG / ShareIt!
Vogelsanger Strasse 78
50823 Koeln
Germany

Phone: +49-221-2407279
Fax: +49-221-2407278
E-Mail: service@shareit.com

US and Canadian customers may also order by calling 1-800-903-4152.

(Orders only please! We cannot provide any technical information about the program.)

US check and cash orders can be sent to our US office at:

ShareIt! Inc.
P.O. Box 844
Greensburg, PA 15601, USA

Tel. (724) 850-8186
Fax. (724) 850-8187

THE ABOVE NUMBERS ARE FOR ORDERS ONLY.
THE AUTHOR OF THIS PROGRAM CANNOT BE REACHED AT THESE NUMBERS.

Any questions about the status of the registration options, product details, technical support, volume discounts, dealer pricing, site licenses, etc, must be directed to zgonnik@math.dvgu.ru.

On payment approval (usually, in one-two business days), We'll send you the full version of the KCompiler by email as soon as possible.

If you will not get your registered version within a reasonable amount of time (three business days for credit card payments or two weeks for other payments), please notify us about that! We're very sorry for any inconvenience caused by those delays.

Important: when filling the order/registration form, please verify that your e-mail address is correct. If it will not, we'll be unable to send you the full version.

1.7. Limitations of demo version

- copyright message box appears when formula is being compiled
- only up to 10 functions may be added

- only up to 2 variables may be added
- only ctBuilt_in function type is available
- AutoConfigure method is not available

1.8. Versions history

KCompiler v.1.20

- Built-in functions were removed – any external function can be now the same as the built-in one

KCompiler v.1.11

- Optimizer was changed a few
- Compilation process became faster, KCompiler's size became smaller

KCompiler v.1.10

- KCompiler now can rearrange operations order and replace them to achieve greater performance
- Operations now can be executed in parallel by all existing Intel and AMD processors produced since 1995
- All integer types added
- Functions may be called faster if they use less then 8 FPU data registers
- New mode of passing parameters to functions added

KCompiler v.1.01

- Some bugs fixed
- Floating-point constants now use as small area of memory as possible to contain them
- Data alignment added

KCompiler v.1.00

- At last I have realized my project! Generated code is as fast as Delphi's :(

KCompiler v.0.8, 0.9

- New types of functions added
- Some bugs fixed

KCompiler v.0.7

- This is the first really working version of KCompiler. Only 70% of project is realized, but it is working!

1.9. About

KCompiler

E-Mail: zgonnik@math.dvgu.ru
Web-Page: http://www.dominsoft.narod.ru

2. Common guidelines

This chapter will explain you how to use KCompiler and will tell you about the data that KCompiler operates.

2.1. What is the formula

The formula that KCompiler operates is an ANSI string (AnsiString Delphi type). Its format may be described in EBNF as the following:

Formula ::= Term { ('+' | '-') Term }
Term ::= Factor { ('*' | '/') Factor}
Factor ::= Real number | Variable | Function | '(' Formula ')'
Real number ::= Number | Number ('e' | 'E') Integer number
Number ::= Integer number | Integer number '.' Unsigned integer
Integer number ::= Unsigned integer | ('+' | '-') Unsigned integer
Unsigned integer ::= Digit | Digit Unsigned integer
Digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Variable ::= Identifier
Identifier ::= First symbol | First symbol IdEnd
First symbol ::= '_' | 'a' .. 'z' | 'A' .. 'Z'
IdEnd ::= Symbol | Symbol IdEnd
Symbol ::= '_' | 'a' .. 'z' | 'A' .. 'Z' | '0' .. '9'
Function ::= Identifier | Identifier '(' Actual parameters ')'
Actual parameters ::= Formula | Formula ',' Actual parameters

Note1: formulas with integer formal parameters have serious limitation described in section "**5.3. Parameters passing**".

Note2: KCompiler doesn't make differences between uppercase and lowercase letters, i.e. it is case-insensitive. Any number of spaces and control characters symbols may be added – they will be removed by KCompiler in the right way (i.e. if formula with spaces contains syntax errors, but these errors disappear while removing spaces, these errors still stay).

2.2. Compiling expression

There probably two ways to get machine code representation of given formula:

1) Set AutoCompile property to TRUE and assign ExprString property to required formula

Example:
{… your code}
KCompiler1.AutoCompile := true;

KCompiler1.ExprString := ‘x+4’;
{… your code}

2) Assign ExprString property to required formula and call Compile method

Example:
{… your code}
KCompiler1.ExprString := ‘x+4’;
KCompiler1.Compile;
{… your code}

2.3. Evaluating expression

To know formula’s result call Evaluate method, take sure that formula was compiled without errors.

Example:
{… your code}
if KCompiler1.CompError = eNone then Edit1.Text := FloatToStr(KCompiler1.Evaluate)
else Edit1.Text := ‘Incorrect expression!’;
{… your code}

2.4. Syntax errors handling

KCompiler mirrors compilation status to read-only CompError property.

2.4.1. CompError property

Type

TError

Declaration

TError = (eNone, eDomain, eNumFormat, eValExpected, eOpExpected, eEdge, eUnknown, eOutOfRange, eType, eVarParameter, eParameter);

Description

Error types:
eNone – compilation was successful
Example: 5+4*2-cos(x)

eDomain - error by constants calculating
Example: 5/0; ln(-5)

eNumFormat - invalid numeric format
Example: 5.5.2

eValExpected - value expected
Example: 5+

eOpExpected - operation expected
Example: 5x; 4(2)

eEdge - ')' missing
Example: (4+2

eUnknown - unknown identifier
Example: 5+unknown*2

eOutOfRange - Integer constant is out of range
Example: 11111111111111111111111111111111111

eType - incompatible types or not supported by this version of KCompiler
Example: pow(x, y), where pow(x: extended; a: byte): extended, and x, y: double - y is incorrect

eVarParameter - var parameter is invalid
Example: pow(x, 4), where pow(var x: extended; a: byte): extended, and x: double

eParameter - number of parameters greater or less than required
Example: pow(5, 2, 4), where pow(x: extended; a: byte): extended

Note: with eDomain…eParameter errors machine code consist of simple “ret” instruction

Effect

Lets your program know compilation status and output status messages to user

2.5. Coding guidelines

This chapter contains list of differences between code generated by Delphi and KCompiler which may not be changed via options or other ways:

1) ESP-addressing is used to access intermediate values in memory, so all functions should change ESP register correctly due to their types (all Delphi functions support this)
2) “push val” instruction is not used – “sub ESP, imm” and “mov [ESP], val” are used instead for parameters passing. This variant takes more space but not slower (and may be faster) for all existing pipelined processors
3) if actual parameter’s type matches formal parameter’s type and may be unary minus operation is applied than value will be passed avoiding using FPU registers
4) if integer format is not supported by FPU the value will be converted to the nearest supported format as early as possible in the code due to avoid “large load after small stores” problem

2.6. KCompiler's speed

All algorithms used in KCompiler have maximum linear difficulty or bounded by very small constants, so the compilation process is fast enough

2.7. Generated code

Generated code is usually faster than code generated by Borland Delphi, Borland C++ and Microsoft Visual C++ up to several times, but it may be slower than Intel's C++ code. However, KCompiler may work with functions faster than compilers, listed above, so if your formula contains a lot of functions written for KCompiler, it may give significant performance improving.

Note: results that were got during testing process are expression- and system-dependent; also listed compilers' purposes differ from KCompiler's purposes, so you shouldn't rely on contents of this chapter in comparing compilers. This opinion is subjective.

3. Configuring KCompiler

This chapter will explain you how to configure KCompiler to achieve greater speed on your formula.

3.1. Auto configuration

KCompiler provides AutoConfigure method to set all settings automatically.

3.1.1. AutoConfigure method

Declaration

procedure AutoConfigure(Config: TAutoConfigure);

Description

Configures KCompiler to achieve better performance with targeting processors family.

Effect

You get fast code without any work.

3.1.2. TAutoConfigure type

Declaration

TAutoConfigure = (acDefault, acAuto, acP5, acP6, acPIV, acAMD);

Description

Selects optimizations for each processor. Anywhere, OutOfOrder and EliminateBrackets properties will be set to ooSoft and ebSoft respectively.

Table 3.1. TAutoConfigure type

Config value	CPU type	SmartLoad	BackOperation	ParallelExecution
acDefault*	P6 family	true	true	true
acAuto	auto	N/A**	N/A**	N/A**
acP5	P5 family	false	true	true
acP6	P6 family	true	true	true
acPIV	PIV family	false	false	false
acAMD	any AMD	true	false	false

* - KCompiler was written using machine powered by Intel Pentium III 800 EB processor, so this CPU type is default

** - CPU type will be detected automatically with “cpuid” instruction and values will be set respectively. If CPU type can’t be recognized, default values will be loaded.

3.2. Expression simplifying

KCompiler has smart mode of expression simplifying. In the common case it calculates all possible constants and values of functions with constant parameters, for example, “2+3*x/2*z*4-cos(0)” will be compiled as “1+6*x*z”. However, this mode is not compatible with Delphi (Delphi simplifies expressions like “4*3/x/2” to “12/x/2”), but it may bring great speed increasing to your application. All constants are kept in the smallest by size floating-point format that can contain them without loosing precision.

Examples:

“2+3*x/2*z*(3+1)-cos(0)” -> “1+6*x*z”

3.3. Formula analyzing

KCompiler provides two options to control formula analyzing: OutOfOrder and EliminateBrackets. They are available with OutOfOrder and EliminateBrackets properties of KCompiler component.

3.3.1. OutOfOrder property

Type

TOutOfOrder

Declaration

TOutOfOrder = (ooStandart, ooSoft, ooAgressive);

Description

Defines execution mode – may bring speed increasing, also reduces number of divisions, which is known to be a very slow instruction.

ooStandart mode - operations are executed by their order in expression

ooSoft mode - +, -, *, / are executed out-of-order

Examples:

a1*a2*a3*a4*a5*a6*a7*a8 -> ((a1*a2)*(a3*a4))*((a5*a6)*(a7*a8))
a1/a2/a3 -> a1/(a2*a3)

ooAgressive - is the same as ooSoft

Effect

As you can see some divisions may be replaced by faster multiplication, and performance will be improved. Also some operations don't depend on results of earlier performed operations, so improving parallelism may be enhanced.

3.3.2. EliminateBrackets property

Type

TEliminateBrackets

Declaration

TEliminateBrackets = (ebStandart, ebSoft, ebAgressive);

Description

Brackets will be eliminated due to mathematical rules.

ebStandart - brackets are not eliminated

ebSoft - brackets are eliminated with +, -, *

Examples:

(a1*a2*a3)*a4 -> a1*a2*a3*a4
a1+(a2+a3)+a4 -> a1+a2+a3+a4
a1*(a2+a3) -> a1*(a2+a3)

ebAgressive - brackets are eliminated with +, -, *, /. You may lose division by zero where it should be - use with care.

Examples:

a1/(a2/a3)/a4 -> a1/a2*a3/a4
a1+(a2+a3)+a4 -> a1+a2+a3+a4
a1*(a2+a3) -> a1*(a2+a3)

Effect

This option lets compiler change order of operations given by user. This may increase effect of OutOfOrder option.

3.4. Optimization set

KCompiler provides three options to control optimization process: BackOperation, ParallelExecution and SmartLoad. They are available with BackOperation, ParallelExecution and SmartLoad properties of KCompiler component.

3.4.1. BackOperation property

Type

TBackOperation

Declaration

TBackOperation = boolean;

Description

Lets KCompiler delay operations execution until FPU stack is full. Operand buried under the top of stack will be extracted with “fxch” instruction that is executed for zero cycles by all Intel processors since Pentium and all AMD processors since Duron. Use this option in pair with OutOfOrder option.

false – all operations are executed immediately after operands loading

true – operation execution is delayed until the stack is full

Effect

This option lets eliminate 2-cycles operation-after-loading penalty with Intel P6 family processors, also it allows use operands those are already in stack avoiding cache hierarchy or memory subsystem. However, using of "fxch" instruction is not so effective with Intel Pentium 4 and AMD Athlon and Duron processors, because retirement bandwidth for these processors is limited and instruction window is large enough, so you may see performance degrading. Use AutoConfigure procedure to set all options automatically.

3.4.2. ParallelExecution property

Type

TParallelExecution

Declaration

TParallelExecution = boolean;

Description

Reorders operations to achieve improving parallelism. Has no effect if BackOperation is not set.

false – operations are executed by their order

true – operations are executed in parallel

Examples:

((a1*a2)*(a3*a4))*((a5*a6)*(a7*a8)) -> will be executed as: b1 = a1*a2, b2 = a3*a4, b3 = a5*a6, b4 = a7*a8; c1 = b1*b2, c2 = b3*b4; result = c1*c2, where b1, b2, b3, b4, c1, c2 are intermediate results.

Effect

This option lets compiler change order of operations execution due to improving parallelism enhancing. Anywhere it may increase "fxch" instruction pressure and degrade performance with Intel Pentium 4 and AMD Athlon and Duron processors.

3.4.3. SmartLoad property

Type

TSmartLoad

Declaration

TSmartLoad = boolean;

Description

Controls source for data.

false – all data will be loaded from FPU stack if possible.

true – all data will be loaded from FPU stack if possible, except aligned single and double precision floating-point values.

Effect

For Intel P6 family processors and AMD Athlon and Duron processors loading aligned data from cache is performed faster than using internal FPU structures up to 2 times, but for Intel Pentium 4 processors FPU structures are used faster.

3.5. Data alignment

KCompiler provides two ways to control data alignment: using AlignMode property and RegisterVarA function.

3.5.1. AlignMode property

Type

TAlignMode

Declaration

TAlignMode = boolean;

Description

Controls data alignment.

false – no data alignment will be performed

true – all constants and intermediate values will be aligned by their natural boundaries:
8-bit data – at any address
16-bit data – to be contained within an aligned four byte word
32-bit data – so that its base address is multiple of four
64-bit data – so that its base address is multiple of eight
80-bit data – so that its base address is multiple of sixteen

Effect

For all x86-compatible processors a misaligned data access can incur significant performance penalties. This is particularly true for cache line splits. However, aligned data usually requires more space, so KCompiler has smart algorithm packing all 32- and 80-bit values into 16-byte blocks, but single 80-bit value will still require 16 bytes of memory.

3.5.2. RegisterVarA function

Declaration

function RegisterVarA(AVariable: TVariable; var NewPlace: pointer): Integer;

Description

This function registers new variable "AVariable" and returns its handle or zero if operation was unsuccessful. Also check for alignment is performed and unaligned variables will be reallocated to aligned place with copying current value. "NewPlace" contains pointer to new aligned variable's position or to old if variable was at aligned location.

Note: variable's copy is contained somewhere in internal structures, so after KCompiler instance has been destroyed, value will be lost.

4. Variables

This chapter contains information that will help you to extend KCompiler's capabilities using variables in expression.

4.1. Information about variable

All information about variable required to use it in formula is contained in structure of TVariable type.

4.1.1. TVariable record

Declaration

```
TVariable = record
  name: string;
  VarType: TVarType;
  VarAddr: pointer;
end;
```

Description

Contains information about variable.

4.1.2. TVariable.name field

Declaration

name: string;

Description

Determines variable's name in formula, it may differ from variable's real name.

4.1.3. TVariable.VarType field

Type

TVarType;

Declaration

TVarType = (vtInt8, vtUInt8, vtInt16, vtUInt16, vtInt32, vtUInt32, vtInt64, vtSingle, vtDouble, vtExtended);

Description

Determines variable's type, all listed types match real Delphi types; matching table is listed below.

Matching table

Table 4.1. Supported types of variables

KCompiler type	Delphi type
vtInt8	ShortInt
vtUInt8	Byte
vtInt16	SmallInt
vtUInt16	Word
vtInt32	Integer
vtUInt32	LongWord
vtInt64	Int64
vtSingle	Single
vtDouble	Double
vtExtended	Extended

4.1.4. TVariable.VarAddr field

Declaration

VarAddr: pointer;

Description

Determines variable's linear address.

4.2. Variable list

KCompiler contains several methods and data structures to add new variables and link them with real variables in your program to provide dynamical variables redress.

4.2.1. VarCode function

Declaration

function VarCode(s: string): Integer;

Description

Returns handle of the variable named by “s” parameter.

4.2.2. RegisterVar function

Declaration

function RegisterVar(AVariable: TVariable): Integer;

Description

This function registers new variable “AVariable” and returns its handle or zero if operation was unsuccessful.

4.2.3. RegisterVarExt function

Declaration

function RegisterVarExt(name: string; VarType: TVarType; VarAddr: pointer): Integer;

Description

This function registers new variable with alias “**name**”, type “**VarType**”, address “**VarAddr**” and returns its handle or zero if operation was unsuccessful.

Note: this function has the same effect as **RergisterVar** function with accordingly filled fields of “AVariable” parameter.

4.2.4. GetVar function

Declaration

function GetVar(var AVariable: TVariable; num: Integer): boolean;

Description

This function gets description of variable with “num” handle and copies it to “AVariable” parameter. If operation was successful returned value will be non-zero (TRUE).

4.2.5. RemoveVar function

Declaration

function RemoveVar(num: Integer): boolean;

Description

This function removes record about variable with “num” handle. If operation was successful returned value will be non-zero (TRUE).

4.2.6. ClearVars procedure

Declaration

procedure ClearVars;

Description

Removes information about ALL registered variables.

4.2.7. FVariables field

Declaration

FVariables: array of TVariable;

Description

Contains information about all registered variables. This field is made “public” due to optimization and simplifying work with registered variables. Use it to copy all information about variables from one instance of KCompiler to another or to display list of registered variables.

Note: never change single elements of this array manually: it may cause an incorrect work of KCompiler.

4.3. Data processing and compatibility with Delphi

All integer data is being processed using FPU unit. It means that there will be no overflow and integer division and modulo operations are not supported; also generated code is slower for addition and subtraction than using ALU unit, but may be faster for division and multiplication, and it differs from the code compiled by Delphi. However, it's fully compatible with mathematical rules and user is better to see 5/2 = 2.5 instead of 5/2 = 2 as using integer division.

Note: integer division and modulo operations may be realized with functions.

5. Functions

This chapter contains information that will help you to extend KCompiler's capabilities using functions in expression.

5.1. Information about function

All information about function required to use it in formula is contained in structure of TFunction type.

5.1.1. TFunction record

Declaration

```
TFunction = record
  name: string;
  CallType: TCallType;
  VarList: TAVarDef;
  FuncResult: TFuncResult;
  StackNeeds: Integer;
  FuncAddr: pointer;
  Simplify: boolean;
  InlinePart: array of byte;
end;
```

Description

Contains information about function.

5.1.2. TFunction.name field

Declaration

```
name: string;
```

Description

Determines function’s name in formula, it may differ from function’s real name.

5.1.3. TFunction.CallType field

Type

TCallType;

Declaration

TCallType = (ctRegister, ctPascal, ctCdecl, ctStdcall, ctBuilt_in);

Description

Determines function’s calling convention.

Matching table

Table 5.1. Supported types of functions

KCompiler convention	Delphi convention
ctRegister	Register
ctPascal	Pascal
ctCdecl	Cdecl
ctStdcall	Stdcall
ctBuilt_in	none

Description

As you can see, KCompiler supports all Delphi function types except Safecall. Also new ctBuilt_in type was added. It allows compiler to pass floating-point parameters using FPU stack, what may greatly increase performance speed, but such function usually should be written manually using assembler or will be distributed with KCompiler component freely or not. See more about this calling convention in chapter “**5.3. Parameters passing**”.

5.1.4. TFunction.VarList field

Type

TAVarDef

Declaration

TAVarDef = array of TVarDef;

Description

Contains information about function parameters. Number of parameters is equal to TAVarDef length.

5.1.5. TVarDef record

Declaration

```
TVarDef = record
  VarType : TVarType;
  PushStyle : TPushStyle;
end;
```

Description

Contains information about one parameter.

5.1.6. TVarDef.VarType field

Description

See description of this field in chapter “**4.1.3. TVariable.VarType field**”.

5.1.7. TVarDef.PushStyle field

Type

TPushStyle

Declaration

TPushStyle = (psValue, psReference);

Description

Defines whether parameter should be passed by value (psValue) or by reference (psReference) – like using “var” in Delphi.

5.1.8. TFunction.FuncResult field

Type

TFuncResult

Declaration

TFuncResult = (frInt8, frUInt8, frInt16, frUInt16, frInt32, frUInt32, frInt64, frSingle, frDouble, frExtended);

Description

Determines function result’s type, all listed types match real Delphi types; matching table is listed below.

Matching table

Table 5.2. Supported types of functions’ results

KCompiler type	Delphi type
frInt8	ShortInt
frUInt8	Byte
frInt16	SmallInt
frUInt16	Word
frInt32	Integer
frUInt32	LongWord
frInt64	Int64
frSingle	Single
frDouble	Double
frExtended	Extended

5.1.9. TFunction.StackNeeds field

Declaration

StackNeeds : Integer;

Description

Determines how many FPU registers function needs to work correctly. Set this field to the least possible value to achieve greater performance, however, set it to eight, if you are not sure. If this value is greater than 8, it will be set to 8, and if it is less than 0 it will be set to 0 automatically.

5.1.10. TFunction.FuncAddr field

Declaration

FuncAddr : Pointer;

Description

Determines function's linear address. If this value is equal to "**nil**", "**InlinePart**" will be used.

5.1.11. TFunction.Simplify field

Declaration

Simplify: boolean;

Description

Determines whether function with constant parameters should be calculated during compilation process.

5.1.12. TFunction.InlinePart field

Declaration

InlinePart: array of byte;

Description

If "**FuncAddr**" is equal to "**nil**", then function's code, containing in the "InlinePart" array will be inserted in KCompiler's code.

5.2. Function list

KCompiler contains several methods and data structures to add new functions and link them with real functions in your program to use any available function in formula.

5.2.1. FuncCode function

Declaration

function FuncCode(s: string): Integer;

Description

Returns handle of the function named by “s” parameter.

5.2.2. RegisterFunc function

Declaration

function RegisterFunc(AFunction: TFunction): Integer;

Description

This function registers new function “AFunction” and returns its handle or zero if operation was unsuccessful.

5.2.3. RegisterFuncExt function

Declaration

function RegisterFuncExt(const name, VarList: string; const CallType: TCallType; const FuncResult: TFuncResult; const StackNeeds: Integer; const FuncAddr: pointer; const Simplify: boolean; InlinePart: array of byte): Integer;

Description

This function registers new function with alias “**name**”, formal parameters “**VarList**”, calling mode “**CallType**”, result type “**FuncResult**”, required FPU stack slots “**StackNeeds**”, address “**FuncAddr**”, simplify mode “**Simplify**”, inline code “**InlinePart**” and returns its handle or zero if operation was unsuccessful. Format of “VarList” string is listed above.

VarList format

"VarList" string consist of 4-characters pieces, each of them determines one parameter's passing style and its type.

Table 5.3. VarList format

r|v	i08|u08|i16|u16|i32|u32|i64|f32|f64|f80

↑ passing style ↑ parameter's type

Table 5.4. Passing style

VarList's first character	KCompiler style	Delphi style
r	psReference	"var" parameter
v	psValue	simple parameter

Table 5.5. Parameter's type

VarList's 2-4 characters	KCompiler type	Delphi type
i08	vtInt8	ShortInt
u08	vtUInt8	Byte
i16	vtInt16	SmallInt
u16	vtUInt16	Word
i32	vtInt32	Integer
u32	vtUInt32	LongWord
i64	vtInt64	Int64
f32	vtSingle	Single
f64	vtDouble	Double
f80	vtExtended	Extended

Example:
For function SomeFunc(x, y: double; var k, f: Integer; b: Single; v: Int64; c, a: LongWord): byte;
VarList = 'vf64vf64ri32ri32vf32vi64vu32vu32'.

5.2.4. GetFunc function

Declaration

function GetFunc(var AFunction: TFunction; num: Integer): boolean;

Description

This function gets description of function with "num" handle and copies it to "AFunction" parameter. If operation was successful returned value will be non-zero (TRUE).

5.2.5. RemoveFunc function

Declaration

function RemoveFunc(num: Integer): boolean;

Description

This function removes record about function with “num” handle. If operation was successful returned value will be non-zero (TRUE).

5.2.6. ClearFuncs procedure

Declaration

procedure ClearFuncs;

Description

Removes information about ALL registered functions.

5.2.7. FFunctions field

Declaration

FFunctions: array of TFunction;

Description

Contains information about all registered functions. This field is made “public” due to optimization and simplifying work with registered functions. Use it to copy all information about functions from one instance of KCompiler to another or to display list of registered functions.

Note: never change single elements of this array manually: it may cause an incorrect work of KCompiler.

5.3. Parameters passing

Information about function parameters and calling mode should be passed to KCompiler through TFunction record, using RegisterFunc function, with CallType (see chapter "CallType field") and VarDef fields.

5.3.1. Differences between parameters passing

Table 5.6. Parameters passing

Calling type	Parameter order	Clean-up	GP registers	FPU registers
ctRegister	Left-to-right	Routine	Yes	No
ctPascal	Left-to-right	Routine	No	No
ctCdecl	Right-to-left	Caller	No	No
ctStdCall	Right-to-left	Routine	No	No
ctBuilt_in	Left-to-right*	Routine	Yes	Yes

*With ctBuilt_in type parameters in FPU registers will be passed Right-to-left

5.3.2. Passing parameters with ctBuilt_in type

The main kind of ctBuilt_in call mode is possibility to pass as much as possible parameters using registers. It means that you will be able to create functions as fast as built-ins. With integer parameters it works as ctRegister call type, but additionally floating-point parameters can be passed through FPU stack.

Number of register floating-point parameters

The number of floating-point parameters passed through FPU registers depends on StackNeeds field of TFunction structure and is calculated as REG_PARS = min(FLOAT_PARS, 8-StackNeeds). In this case StackNeeds field means how many additional FPU stack slots your function requires, so total number of FPU stack slots your function may use is calculated as TOTAL_SLOTS = REG_PARS+StackNeeds.

Parameters allocation in FPU registers

All register floating-point parameters with ctBuilt_in functions' type are passed from right to left. It means that the first register parameter will be deeper in the FPU stack then the last one. After function's work all parameters should be **pulled** out of FPU stack.

Example:

function SomeFunc(x, y, z, w : double; var k, f: Integer; b: Single; v: Int64; c, a: Integer): byte;
StackNeeds = 6;

Table 5.7. Parameters passing example

Parameter's name	Allocation
x	ST(1)

y	ST(0)
z	stack
w	stack
k address	EAX
f address	EDX
b	stack
v	stack
c	ECX
a	stack

5.3.3. Compatibility with Delphi

Besides floating-point parameters KCompiler supports integer parameters passing, but this mode has serious limitations: only unary minus operation, performed in ALU unit, is supported. If you wish to pass integer value as floating-point parameter, it will be processed using FPU unit. It means that there will be no overflow and integer divisions and modulo operations are not supported; also generated code is slower for addition and subtraction than using ALU unit, but may be faster for division and multiplication, and it differs from the code compiled by Delphi. However, it's fully compatible with mathematical rules and user is better to see 5/2 = 2.5 instead of 5/2 = 2 as using integer division.

Note: integer division and modulo operations may be realized with functions.

5.3.4. Routines calling methods

This chapter describes common conventions, using by routine's call.

Results

KCompiler expects to get routine's results where standard Delphi function should return them.

Table 5.8. Routines' results

Result type	Allocation
vtInt8	AL
vtUInt8	AL
vtInt16	AX
vtUInt16	AX
vtInt32	EAX
vtUInt32	EAX
vtInt64	EDX:EAX
vtSingle	ST(0)
vtDouble	ST(0)
vtExtended	ST(0)

Note: if routine's result is vtSingle, vtDouble or vtExtended then it always may use one FPU register.

Calling routines

All routines, called during expression's evaluating, should obey several simple rules:

1) EBP, EBX, ESI, and EDI registers must be preserved (all Delphi routines)
2) Function mustn't use more FPU registers than it has required through NeedStack field
3) Result should be returned as described in "**Results**" section (all Delphi routines)
4) Floating-point parameters passed through FPU registers should be **pulled** out, for example, using "fstp st(0)" instruction several times
5) Stack should be cleared due to function type (all Delphi routines)
6) Inline function shouldn't contain relative "jmp" or "call" instructions targeting outside the bounds of function

Calling routines, KCompiler uses several conventions:

1) Some parameters can be passed in registers as described above
2) Inline function gets parameters as any other function. Note that there will be no return address for inline function, so the last stack parameter will be available on [ESP] address.
3) Routine may freely modify EAX, ECX and EDX registers
4) If function has only constant parameters and it's "**Simplify**" field is set to TRUE, it will be evaluated during compilation
5) If floating-point parameter should be passed through memory and it consist of simple variable, it's negation or a constant, it will be passed avoiding FPU registers
6) If inline function's body contains "ret" instruction, it may crash the program. Consistence of FPU and other registers will be undefined

5.4. Built-in functions

Since version 1.20 KCompiler has no built-in functions. It means that any user's function may have the same behavior as earlier built-ins. However, you can use highly optimized functions containing in free "KLib.pas" library.

Appendix A. Set of types

This chapter contains all types that you will need working with KCompiler and which are not predefined in Delphi.

Listing A.1. Set of types

```
TError = (eNone, eDomain, eNumFormat, eValExpected, eOpExpected, eEdge, eUnknown,
eOutOfRange, eType, eVarParameter, eParameter);

TPushStyle = (psValue, psReference);

TVarType = (vtInt8, vtUInt8, vtInt16, vtUInt16, vtInt32, vtUInt32, vtInt64, vtSingle, vtDouble,
vtExtended);

TVarDef = record
  VarType: TVarType;
  PushStyle: TPushStyle;
end;
```

```
TAVarDef = array of TVarDef;

TCallType = (ctRegister, ctPascal, ctCdecl, ctStdcall, ctBuilt_in);

TFuncResult = (frInt8, frUInt8, frInt16, frUInt16, frInt32, frUInt32, frInt64, frSingle, frDouble,
frExtended);

TFunction = record
  name: string;
  CallType: TCallType;
  VarList: TAVarDef;
  FuncResult: TFuncResult;
  StackNeeds: Integer;
  FuncAddr: pointer;
  InlinePart: array of byte;
end;

TVariable = record
  name: string;
  VarType: TVarType;
  VarAddr: pointer;
end;

TTrigMode = (tmSafe, tmNothing, tmDelphi);

TOutOfOrder = (ooStandart, ooSoft, ooAgressive);

TEliminateBrackets = (ebStandart, ebSoft, ebAgressive);

TAlignMode = boolean;

TBackOperation = boolean;

TParallelExecution = boolean;
```

Appendix B. Advanced syntax errors handling

To handle syntax errors in formula using Delphi exceptions, you may include the following unit:

Listing B.1. Syntax errors handling using Delphi exceptions (KCompilerEx.pas)

```
unit KCompilerEx;

interface
uses
  KCompiler, SysUtils;

type
  TKCompilerEx = class(TKCompiler)
    procedure Compile; override;
```

```
  end;
  ECompError = class(Exception);

var
  Msgs : array [TError] of string =
  ('Compiled succesfully!',
   'Constant value cannot be represented in current format!',
   'Invalid numeric format!',
   'Value expected!',
   'Operation expected!',
   '")" expected!',
   'Uknown indefitier!',
   'Constant is out of range!',
   'Incompatible types!',
   'Types of formal and actual var parameters must be identical!',
   'Number of parameters doesn''t match function prototype!');

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('DOMIN', [TKCompilerEx]);
end;

{ TKCompilerEx }

procedure TKCompilerEx.Compile;
begin
  inherited;
  if CompError <> eNone then
    raise ECompError.Create(Msgs[CompError]);
end;

end.
```

Legal notices

Partnership

I will appreciate all proposals and suggestions. If you will have any business or other proposals please feel free to mail me.

If you have any questions about getting source code of described product, please send all your offers to zgonnik@math.dvgu.ru