Handling of the arithmetic expressions

The text structure of one formula or calculation is the following.

```
GRAPHIK:
ID=>3
NAME=>"Demo"
DATEI=>"Grphexmpl2.tmp"
PARAMETER=>3
RESOLUTIONS=>1 0.5 0.75 0
FORMEL=>
if (p_0 neq 3) { Fehler } else { p_1 + p_2 - p_3 }
```

First comes the identification number of the formula. The user interface enables the user to select such an id within the existing set of values, and will go through the list and execute each code with the same ID. Therefore several calculations
Then a name is given (free value), and eventually a file name (both are unused in our case) that was thought to hand over a description of the page in which the values are presented.
The next parameter states the number of parameters (used under the name p_ in the code) used by the code. This value is passed by the user interface into parameter p_0.
The resolution field states the step values with which the values of the parameter p_x are incremented or decremented using the + or - keys. The values for all 4 parameters must be present. Those values are only used if the macro SPRACHE is not defined. Otherwise, another scheme is used to set the values of the parameters. First the parameter has to be select with the values 1 to 4. Then the integer part is set with the keys 0-9 and -. The type the . key and set the fractional part with keys 0-9.
The formal field specifies the code compute the value. If one graphic identification number is assigned to more than one formula or calculation, the result values are calculated and displayed on the console in the order in with they are listed in the text file.

At the start of the input file, several values have to be specified for the size of stack and buffers.

```
SPEICHER_KODE:1000
SPEICHER_STACK:200
SPEICHER_CONST:20
```

Those are in the order:
temporary buffer in which the code of the formula/calculation are stored when read in from the file. This code is then copied into an appropriate memory space.
the size of the argument stack that is used when executing the calculations. Instruction and function take their arguments from this stack and put their result on it to make them available to the next instruction or function.
temporary buffer for the constant values. Only used if compiler switch ARG_DOUBLE is defined. Those values are then copied into an appropriate memory space.

For instance, 20 graphic formulas can be stored. This is defined by the MAXGRAPHS compiler define in file Parser.h. Later versions may take it from the text input file.

For the formula / calculations, different control structures are provided

if (...) { .... } else { ... }
<Variable Name> = ... ;
When parsing the formula, the variables are stored in a table. This table will be called variable table in the following document. When the code is executed, the value calculated between the following the = is assigned to this variable. When a variable is defined the first time a value has to be assigned.
Per default a variable named "Fehler" is defined, and its value set to NAN (Not A Number). If an operation returns this value, the execution is stopped and a message is displayed in the console.
while (…) { … }
do { … } while (…)
for (<index name>= ..; <index name> <><=>= .. ; ++/--<index name>--/++) { … }

The last ; before the } character ending a control structure path is optional as the parser will end the control path anyway when finding the } character.

Input parameters for the calculation

In total 5 input parameters are handled by the user interface (the underlying software loading and executing the calculations can handle more). The first one p_0 is set to the number of parameters given in the field PARAMETER.
The 4 other input parameters name p_1 to p_4 can be set by the user to any floating point value using the two input schemes described above.

Storage of the calculation formulas

The calculations are stored in form of a list of references to standard routines that are executed one after the other in a loop. In the following document we will call this list Code Table.
Those standard routines take their argument(s) off the stack (Operand Stack) and put their result back on it for use by the next operations (like the Hewlet Packard pocket calculators). This makes it possible by converting the expressions in Reverse Polish Notation or Postfix Notation to generate such a sequential execution list. In this Reverse Polish Notation, the arguments come up first, followed by the operators in decreasing priority order so that they are just executed from left to right.
Those standard routines are defined in files Calculs.h/c. In fact all standard operators and functions have been rewritten to use the operand stack to transmit values.
The conversion to postfix notation is performed when reading in the textual formulas by the routines defined in files Parser.h/c.

For the operands and parameter 2 additional routines have been defined:
put: this routine takes a floating point either from the Code Table at the position just after the executed operation (that must be a reference to itself) or if compiler switch ARG_DOUBLE is defined, takes the integer value there as an index into an array of double values where the constants have been stored, and puts the floating point onto the argument stack for used by the next operations.
read:the input parameters are stored into an array. When the code is executed, the first one is set to the number of parameters (flied PARAMETER) and can be used with the name p_0. This routine reads the integer values just after the presently executing operation (that must be a reference to this routine) and uses it as the index in the array of input parameters and puts the value there onto the argument stack for used by the next operations.

A Graphical description of the principle of operation is given below:

for example to put a value of 2.5 onto the operand stack or the value of the third input parameter, we would find in the code table:

```
Code :               Code:                              Constants Array   Code Table
ARG_DOUBLE not       ARG_DOUBLE
defined              defined

put                  put                                                  read
2.5                  4        ---------------------->   2.5               3            ⇒ aktDaten[3]

```

For this the execution index in the code table has to be incremented once additionally to the incrementation in the execution loop in order to skip the following value.

For the variables, an array (Variables Array) is stored with each formula if compiler switch MIT_VAR is defined:
This array contains the two following column Name and value:

char nom[20]	double valeur

To assign a value to a variable. the **stovar** routine is provided. This routine uses the value following its reference in the code table as the index in the variables table to sure the variables value. This index value is defined when parsing in the textual formulas. Stovar then takes the last value off the argument stack and stores it into the value tag at the given index in the variables array.
Reading a variables value works the same way with the routine **getvar**. This one reads the value of the value tag at the given index, and deposits it at the top of the arguments stack.
Both routines increment the execution index additionally to the incrementation in the execution loop (routine Execute in Calculs.c) to skip the index value following their reference. For example, a variable stored at index 3 in the variables array:

```
Variable setting code                       Variables array                            Variable reading code

stovar                                                                                 getvar
3          --------------------------------> <name>   <value>   <----------------------------------  3

```

the modifications in the operands stack and variables array are as follows:

```
Operand stack                               Variables    array                                Operand stack
                                            0  Fehler    Failure value (NAN)
top value Y (taken off)  -------------+     1                                  +------------> Y (added)
                                      |     2                                  |
                                      +---> 3  Var index 3       Y        --------------+
```

Implementation of the if else structures:

For this the routines **branchif** and **branchelse** have been defined.
Branchif reads the two integer values following the reference to it in the code table, the first is interpreted as the index of the first element of the else branch (do not forget the after the call to this routine, the execution index will be incremented once (see Execute routine in Calculs.c), and the second as the index of the last element of the else branch in the code array. The first element in the else branch if present is always a reference on the **branchelse** routine.
When called, **branchif** reads the last argument of the operands stack. If it is true (nonzero), then the indexes following its reference are put onto the if else jump stack. Then the code of the if branch is executed until the **branchelse** routine is called that will modify the execution index (with the value in the jump stack) to put it on the last element of the else branch, so that when processing the next execution loop iteration, the else branch has been skipped.
If the last value on the operand stack is false, **branchif** sets the execution index to point on the **branchelse** reference in the code array, so that the next iteration of the execution loop starts at the first operation of the else branch from which the execution runs sequentially so that no more jump has to be considered.

```

                         branchif
+-----------------       index else
|               .        index end                          ----------+
|               .                                                     |
|               .                                                     |
+---------------->       branchelse                                   |
                         ...                                          |
                         Last element of the else branch    <--------+
                         Operation after the else branch
```

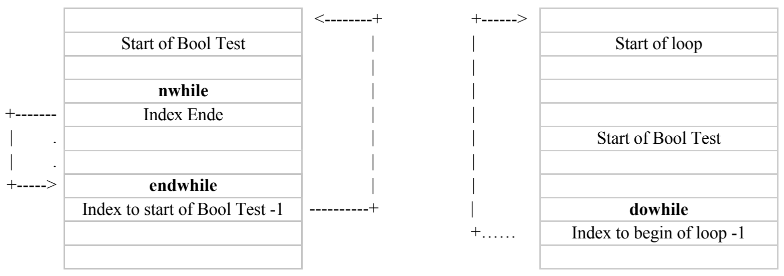
In the boolean test expressions different operations more common to Perl script language are available:

Reserved word	for	in C
eq	equal	==
neq	not equal	!=
gth	greater then	>
lth	less than	<
geth	greater or equal to	>=
leth	less or equal to	<=
not	not	!

So the end index points on the last element of the else branch, an the else index points to the first element of the else branch that is a reference to the **branchelse** routine, As the execution index will be incremented once more in the execution loop after the jump, the next operation is called properly.

Implementation of the loop control structures:

We start with the description of the while and do while loops. As for the if else structures, the jump indexes always point to the element just before the operation that will have to be executed after the jump. New routines have also been defined that take the result of the test expression of the operand stack, and expect the jump index(s) just after the reference to the routine in the code array.



The loop test condition also produces a value on the operand stack where the test routine takes its input, and expect the jump indexes just after the reference to them in the code array.

The syntax of such loop structures is the following:

while (Expression with lth leth gth geth evtl. + - ...) { .... }
do { .... } while (Expression with lth leth gth geth evtl. + - ...)

For the for loops two additional routines have been defined in order to increment or decrement the value of a variable directly in the variables table to prevent a loading onto the operand stack, the modification of this value and a write back into the variables table. Those are **incVar** and **decVar**, who expect the index of the variable in the variables table just after the reference to them in the code array. In the code array such loops are store as while loops. A graphical due of the implementation is given below:

put
Start value of loop index
stovar
Index in variable table of loop index
Start of Bool Stop Test
nwhile
Index End
incVar
Index in variable table of loop index
endwhile
Index to start of Bool Test -1
Operation ofter loop

In this case we have the implementation of for(Index=Startvaluet;Stop Test;Index++) { ... }. The reference to **incVar** may also be placed at the beginning of the loop if the last operation is a pre incrementation or decrementation. Then we have for(Index=Startwert;Stop Test;++Index).

In case of syntax error:

In case of detection of a syntax error, the parser calls ExecError (in Calculs.c) that replaces the so far generated code by a piece of code that return a NAN value.

Usage of the module:

Before the parsing routine is called, the routines InitCalc (Calculs,c) and InitParse (Parser,c) have to be called. The first one allocates memory for a code array (size given in field SPEICHER_KODE) that is used a temporary buffer when parsing in a formula (when finished the content is copied into a definitive memory allocation of exact size). for the operands stack (size given in field SPEICHER_STACK) and if compiler switch ARG_DOUBLE is defined, a memory allocation (size given in SPEICHER_CONST) used as temporary buffer for the variables of the currently read in calculations. The content is also copied int a find memory location associated with the formula.
The second allocates a memory buffer in which the string of the FORMEL field will be stored to be parsed, if compiler switch DEBUG is defined a text buffer in which a log of the parsing is written, a stack that is used to convert the arithmetic expressions in postfix notation and finally opened the text file to be read in.

Once the parsing in is finished, all the temporary buffers can be released and the file closed. This is done in the Quit routine (Parser,c). The CleanUp routine (Parser,c) is provided to release the memory when ending the application.