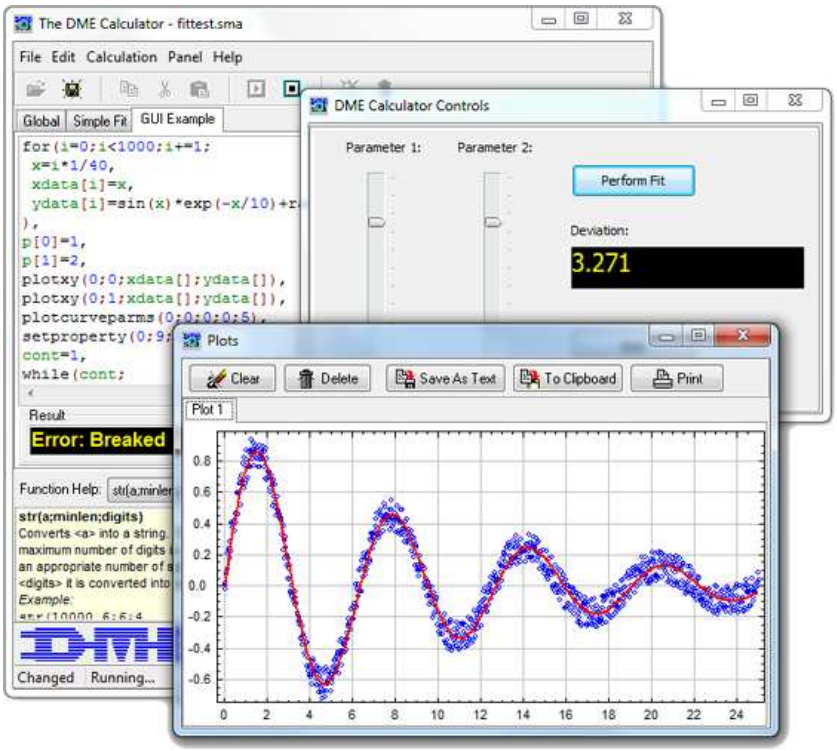
THE *DME Calculator*



DME - Danish Micro Engineering A/S
Transformervej 12 · DK-2730 Herlev · Denmark
Tel: +45 44 84 92 11 · Fax: +45 44 84 91 97
http://www.dme-spm.com · e-mail: dme@dme-spm.dk

Contents

1 Using the *DME Calculator* **1**
- 1.1 Introduction . . . 1
- 1.2 Getting started . . . 3
- 1.3 Example programs . . . 4
  - 1.3.1 Understanding basics . . . 4
  - 1.3.2 Controlling the program flow . . . 8
  - 1.3.3 Defining own functions . . . 10
  - 1.3.4 Text output window . . . 12
  - 1.3.5 Two dimensional plots . . . 13
  - 1.3.6 Further functions . . . 14

2 Creating and Using *DME Calculator* GUIs **15**
- 2.1 Overview . . . 15
- 2.2 Creating GUIs . . . 16
- 2.3 Using GUIs . . . 18
  - 2.3.1 Communication with GUI controls . . . 19

3 Language reference **21**
- 3.1 Syntax . . . 21
- 3.2 Operators . . . 22
- 3.3 Variables . . . 23
  - 3.3.1 Global Variables . . . 24
- 3.4 Comments . . . 25
- 3.5 Functions . . . 25
  - 3.5.1 Program flow functions . . . 25
  - 3.5.2 User interface functions . . . 27

Chapter 1

Using the *DME Calculator*

1.1 Introduction

The *DME Calculator* is a free, advanced calculator for windows. Basically it can be used like a normal pocket calculator to perform simple operations like adding numbers etc. with the advantage that all numbers and operations that have been entered can later be checked and corrected without entering all following data again. Like a normal pocket calculator, the *DME Calculator* has a predefined set of mathematical functions like trigonometric and exponential functions, etc.

Besides the basic mathematical functions the *DME Calculator* also supports functions like if(), for(), while(), etc., turning *DME Calculator*'s mathematical expressions into a real programming language. And as with other programming languages, the *DME Calculator* can calculate with variables which can contain either numbers, arrays of numbers or strings. It is also possible to define own functions and use them in the following expressions.

Besides these language features, the *DME Calculator* has a text and a graphic output window allowing the user to display messages or evaluation results and make two dimentional graphic plots. Special functions allow reading and generating numerical data files as well as communication via the PC's serial port(s).

Since version 3, the *DME Calculator* allows to create graphical user interfaces (GUIs) with buttons, scroll bars and input elements etc. to allow user interaction without having to change the calculator program. To create the GUIs, the *DME Calculator* is equipped with an own user interface editor. The files created with this editor can then be loaded by an *DME Calculator* program, which then can react on user events. The GUI files are simple text files, which, for special purposes can also be created by other programs or even by *DME Calculator* programs. This allows dynamical changes of user interfaces, even while a *DME Calculator* program is running.

Also since version 3, a function for general nonlinear least square fits *fit()* has been

added to provide easy modeling of data. Another added function is the fast fourier transform (FFT).

In summary, the *DME Calculator* is a powerful calculator, integrating many aspects of a programming language. The *DME Calculator* is distributed as freeware and the newest version can always be obtained from DME - Danish Micro Engineering A/S, `http://www.dme-spm.com` or DME Nanotechnologie GmbH, `http//www.dme-spm.de`.

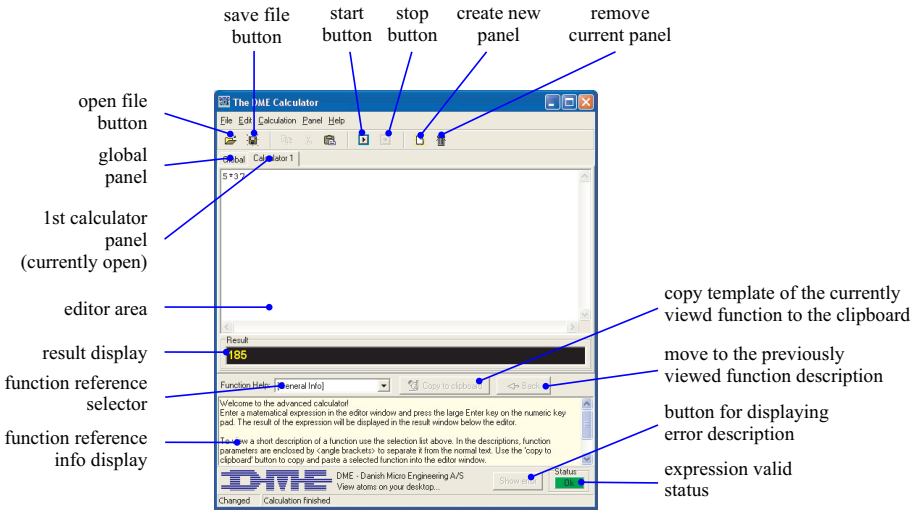


Figure 1.1: The main screen of the *DME Calculator*.

1.2 Getting started

The *DME Calculator* comes as a single .exe file. It can be started directly and commonly no installation procedure is necessary. To open this documentation file from the *DME Calculator*'s menu the .pdf-file must reside in the same directory as the .exe file.

Figure 1.1 shows the main screen of the *DME Calculator*. The program has a built-in text editor where the user program is entered. During typing, the syntax of the user program is permanently checked and the validity is indicated in the lower right corner by a green or a red sign. If the 'expression valid status' is red, the 'show error' button can be used to display a message stating the cause of the invalidity.

> *Try this by entering the expression* `1+2` *in the program area. After typing the '1', the status stays green and after entering the plus sign, the status changes to red. Now pressing the* show error *button displays the message 'Expression or identifier expected', as the plus sign requires something on the left and on the right side. After entering the '2', the status changes to green again.*

To load and save programs, use the open and save file buttons or the corresponding file menu. *DME Calculator* programs have the extension '.sma'.

When a program has been entered and the status is green, the start button becomes enabled and the program can be started by using the start button or pressing the F2 key. When a program terminates, the return value is displayed in the result line.

> *If you have tried entering* `1+2` *as mentioned above, now press the start button. The result of 3 should be displayed in the status bar.*

To perform simple calculations it is quite often nice to see the intermediate results at once, like in a normal pocket calculator. For this purpose the large *Enter* key on the numerical key pad acts like pressing the normal return key and starting the calculation at the same time. For example when adding numbers you can press the key sequence

```
2 <Enter>
+5 <Enter>
+3 <Enter>
```

(where <Enter> means: press the Enter key on the numerical key pad) and always see the intermediate results.

One *DME Calculator* file can contain multiple panels. Each panel has its own editor window and only the expression in the currently active panel is evaluated when the start button is pressed. When a new file is created, a 'Global' and a 'Calculator 1' panel is created. The 'Global' panel differs from all other panels: The global panel can not be evaluated directly, but the expression in the global panel is always evaluated before any other panel. The global panel can be used to define global variables or functions that should be available in all other panels.

By using multiple panels one can easily switch between different calculations or start a different calculation without destroying an old calculation. A new panel is created by pressing the 'create new panel' button. Per default, the normal panels are named 'Calculator 1', 'Calculator 2' etc. A special name can be choosen by using the rename function from the panel menu. All panels are stored within one single file together with the global panel.

Below the result window a short description of all built-in functions can be displayed. hen pressing the 'function reference selector' a list of all predefined functions appears and the desired function can be selected. By using the 'Back' button one can go through all the previously viewed function descriptions.

1.3 Example programs

In this sections some program examples are shown, demonstrating the basic principle of *DME Calculator*'s language. For advanced reading, go to the language reference in chapter 3 on page 21.

1.3.1 Understanding basics

The basic operation principle of the *DME Calculator* language is that of a calculator. As a first simple example enter the expression shown in Example 1.3.1

```
(2+5)*sqrt(3)
```

Example 1.3.1: A first simple example

and press the start button. This calculates $(2+5)*\sqrt{3}$ which has the result 12.12.... You can also enter multiple expressions, separated by a comma (','), as shown below:

```
1+1, 2+2, 3*3
```

Example 1.3.2: Multiple expressions separated by a comma

When you press start, all expressions are calculated but only the result of the last one is returned, which is 9 in this case. As the comma is in fact also an operator like the plus sign, it can appear anywhere in the expression, so

```
1+(4,6)
```

Example 1.3.3: The comma is an operator

returns 7, as the result of `4,6` is 6 and then 1 is added. The comma is an operator that just returns the value on the right side. But now try entering the same expression without the brackets.

```
1+4,6
```

Example 1.3.4: Operator precedence

Now the return value is 6. That is because the comma operator has a lower precedence than the plus sign, so `1+4,6` is similar to writing `(1+4),6`. The behaviour of the comma operator is the same like calculating multiplications before additions, like `2*3+5` is the same as `(2*3)+5` and is different from `2*(3+5)`. The comma operator in fact has a lower precedence than all other operators, so it works just as expected like a 'command separator'.

The *DME Calculator* language also contains basic mathematical functions. The result of Example 1.3.5

```
cos(0)
```

Example 1.3.5: Using functions

is 1 and the the result of

```
sin(pi()/2)
```

Example 1.3.6: Using functions

is also 1. Functions are always written with two brackets containing the function arguments. If a function has no arguments like the `pi()` function that just returns π, the brackets must be also written to distinguish the function from a variable.

When the cursor is placed on a built-in function and the F1 key is pressed, the online help display jumps to the description of that function.

Using variables is as easy as using real numbers, so

```
a=3, 4*a
```

Example 1.3.7: Using variables

the example 1.3.6 returns 12, as first the variable 'a' gets the value 3 and then it is multiplied by 4. Variable names are case dependent, so Example 1.3.8

```
Hugo=1, hugo=2, Hugo
```

Example 1.3.8: Variable names are case dependend

returns 1 as the variable `Hugo` is a different variable than `hugo`.
To make it a little bit more complicated, look at the expression in Example 1.3.9

```
b=(a=6)*2,b+a
```

Example 1.3.9: = is an operator

The equal sign is also an operator that returns the value it assigns, so the expression `a=6` not only assigns 6 to *a* but also has the return value 6. So in Example 1.3.9 *b* is assigned 12, while *a* is assigned 6, so the result is 18.

DME Calculator can work not only with numbers but also with strings. Strings are enclosed in single quotes.

```
'Hello'
```

Example 1.3.10: Strings

The example above just returns 'Hello'. One can also perform operations with a string, so

```
'Hello'+' you'
```

Example 1.3.11: String concatenation

returns 'Hello you'. Strings can also be assigned to variables, so

```
a='Hello',
b=' you',
a+b
```

Example 1.3.12: Variables and strings

Example 1.3.12 does the same as Example 1.3.11 and returns 'Hello you'.
Sometimes it is required to put a number behind a string. Try Example 1.3.13,

```
a=5*2,
'The result is '+string(a)+'.'
```

Example 1.3.13: Variables and strings

which returns 'The result is 10.'. The `string()` function is necessary to force *a* to be a string so that the + operator performs a string concatenation and not a numerical addition. Without the `string()` function, the example would just return the number 10, as the string 'The result is ' and '.' cannot be converted into a number and thus is converted to 0. When converting a string into a number the conversion is done from left to right and the conversion is stopped when the first non-numerical character is found. Example 1.3.14

```
'123abc'*2
```

Example 1.3.14: Variables and strings

returns 246, because the first digits could be converted into a numerical value.

1.3.2 Controlling the program flow

We start with conditional evaluation and using the `if(a;b;c)` function. This is a function with 3 arguments. Multiple function arguments are separated with a semicolon in the *DME Calculator*. Example 1.3.15

```
b=5,
if(b<3;a=2;a=3),
a
```

Example 1.3.15: Conditional evaluation

returns 3, because the `if()` function first evaluates the first argument. If this is true, it evaluates the second argument, otherwise it evaluates the third argument. In the example, b is not less than 3, so the third argument is evaluated and a is assigned 3.

The `if()` also returns the value of the expression it evaluates, so

```
if(2<3;5;6)
```

Example 1.3.16: Conditional evaluation

the example above simply returns 5, as the first expression is true. So example 1.3.15 could also be written in the following way:

```
b=5,
a=if(b<3;2;3),
a
```

Example 1.3.17: Conditional evaluation

The third argument must always be present. If it is not used one can put a zero there:

```
b=5,
if(b>10;b=10;0),
b
```

Example 1.3.18: Conditional evaluation

In the example above *b* is set to 10 if it is greater than 10. Otherwise *b* is not changed.

The first expression in the *if* function is evaluated to check whether it is true or false. In the *DME Calculator* language, true and false are also represented as numbers, where 0 means false and any other value means true. Just evaluating Example 1.3.19

```
2<3
```

Example 1.3.19: Conditions

returns 1 and `3<2` would return 0. Also `if(1;3;4)` returns 3 and `if(0;3;4)` returns 4. There are also an *and* (&) and an *or* (|) operator, as well as a not function. So Example 1.3.20

```
not(2<3 & 5>2)
```

Example 1.3.20: Conditions

returns zero.

Another function that is based on a condition is the `while(cond;expr)` function. This function has two arguments. It evaluates the second argument until as long as the first argument is true.

```
a=1,
i=1,
n=4,
while(i<=n;
 a *= i, i += 1),
a
```

Example 1.3.21: The while() function

Example 1.3.21 returns the factorial of *n*. *n* is set to 4 so the return value is 24. Here *a* is multiplied by *i* and then *i* is incremented by one. The loop is stopped when *n* becomes 5.

The same can also be done with the `for(init;cond;inc;expr)` function. This function has four arguments and is also used for creating a loop. It has two more arguments for initializing and counting up a counter variable. Using the `for()` function, Example 1.3.21 can be rewritten as

```
n=4,
for(a=1,i=1; i<=n; i+=1;
    a*=i),
a
```

Example 1.3.22: The for() function

Using the `for()` function makes it easy to create a loop evaluating an expression with a counter variable i counting up from a to b, which is done by `for(i=a;i<=b;i+=1;expr)`.

1.3.3 Defining own functions

Using the special `define()` function it is very simple to define own functions. The first parameter of the `define()` function is the new function name including parameters. The second parameter is the expression that should be evaluated when this function is called. As with all expressions in the *DME Calculator*, this expression can be very big and complex.

```
define(f(x); 2*x),
f(5)
```

Example 1.3.23: Defining own functions

In the example above a function `f(x)` is defined and then `f(5)` is evaluated which returns 10. The newly defined function is available at once, even in the expression defining the function (see later example 1.3.25) If the function is defined in the global panel, it can be used in all other panels.

All normal variables, i.e. variables not starting with a $-sign (see section 3.3.3 on page 25), that are used and defined within a function definition are local variables. Variables inside the function definition have nothing to do with variables outside the function definition:

```
define(f(x);2*x),
x=3,
f(4)+x
```

Example 1.3.24: Variables within function definitions are local

In the example a variable x is used inside the function definition as well as in the main expression. The return value of this example is 11 and this shows that the variable x out-

side the function definition is a different variable than the x used inside the `define()`. Even after the evaluation of `f(4)` x still has the value 3.

In fact the *DME Calculator* implements a real 'variable-stack' which distinguishes variables even if a function is called within its own defining expression. This allows programming of recursive expressions:

```
define(factorial(n);
 if( n<=1; 1; n * factorial(n-1))
),
factorial(5)
```

Example 1.3.25: Recursive expressions

Here the function `factorial(n)` calculates the factorial of the integer n. This is done with a recursive expression: When the function is called with a parameter > 1 the function calls itself again with the parameter decremented by 1. The result of this is multiplied by the original parameter. When the parameter is <= 1, the function simply returns 1. This 'stop-condition' is very important, otherwise this would result in an infinite recursion. The result of `factorial(5)` is $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

Arrays are passed 'by reference': When the contents of an array parameter within a user-defined function is changed, the array in the calling expression is also changed. The next example uses this feature:

```
define(multi(a[];v);
 foreach(i;a[];a[i] *= v)
),
for (i=0;i<10;i+=1;
 anarray[i] = i
),
multi(anarray[];20),
foreach(i;anarray[];
 println(anarray[i])
)
```

Example 1.3.26: Arrays as parameters

Here a function `multi()` is defined, that takes an array as first parameter and multiplies all elements in the array by the second parameter. In the example an array is created and then multiplied by 20.

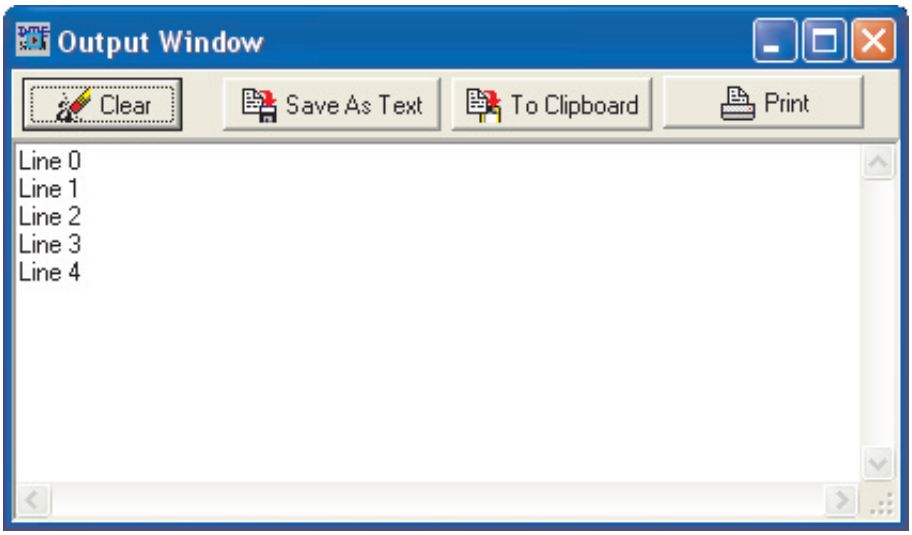


Figure 1.2: A text output example.

1.3.4 Text output window

Sometimes it is necessary to view intermediate results or generate messages during the calculation. For this purpose the *DME Calculator* has two functions called `print()` and `println()`. These functions have a single scalar variable as parameter and write the contents of the variable to a text output window. The following example generates text output:

```
for(i=0; i<5; i+=1;
 println('Line '+string(i))
)
```

Example 1.3.27: Writing text to the text output window

The output of this example is shown in Fig. 1.2. The difference between the functions `println()` and `print()` is that `println()` automatically appends a linefeed. If the `print()` function would have been used in the example the text output would look like 'Line 0Line 1Line2...'.

By means of the buttons of the text output window its contents can be stored into a file, printed, or copied to the clipboard.

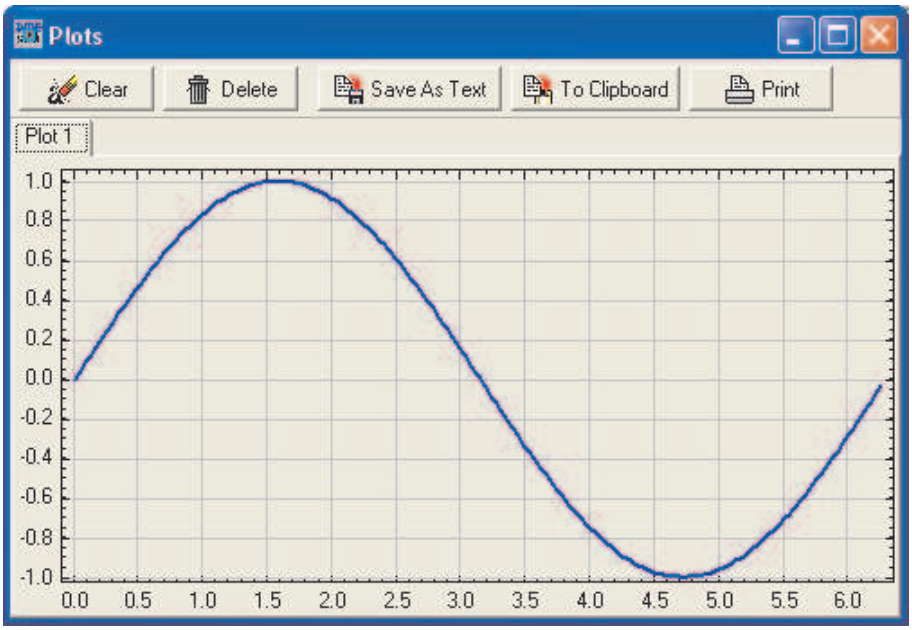


Figure 1.3: The two dimensional graphic window

1.3.5 Two dimensional plots

The *DME Calculator* provides multiple functions for generating and handling two dimensional plots. All function names start with 'plot...' so they can easily be identified. The single line

```
plot(0;0;x;sin(x);0;2*pi();200)
```

Example 1.3.28: Simple two dimensional plot

generates the output shown in Fig. 1.3. The function `plot(tabno;curveno;var;func;start;stop;count)` evaluates *func* and generates *count* data points by setting *var* to values in the interval $[start, stop]$. The two dimensional output window can contain multiple tabs, and on each tab multiple curves can be plotted into the same diagram.

Several functions are available for specifying axis labels, tab label, curve colour and thickness, as well as selecting whether data points should be drawn connected or as a line.

Interactive zooming is supported by dragging a rectangle using the left mouse button. To go back to the automatically scaled view, right-click somewhere on the graphic window. Using the buttons on the top of the window, the current diagram can be copied to the clipboard, saved as ASCII data, or printed.

To plot point data from arrays use the plotxy() function!

1.3.6 Further functions

A complete list of functions is contained in the online help. Especially to mention are the functions *fit()* for performing nonlinear least squares fit by a modified levenberg-marquardt algorithm and *fft()* and *invfft()* for performing fast fourier transform calculations. The *DME Calculator* is also capable of performing file io, starting external programs, doing serial communication and communication via GPIB (IEEE-488).

Chapter 2

Creating and Using *DME Calculator* GUIs

2.1 Overview

Starting from version 3, the *DME Calculator* supports creating user defined graphical user interfaces (GUIs). One can create buttons, edit fields, scrollbars etc. to provide input data for *DME Calculator* programs as well as controlling program behaviour or even fully interactive programs. An example of such a control window is given in Fig. 2.1 showing all basic controls. *DME Calculator* supports the following types of controls:

Button Control The button control creates a button which can be pressed and held down. The button's text can be updated by the *DME Calculator* program.

Label Control The label control's main purpose is to display descriptive text. Even though labels commonly display always the same text, the text of a label can also be changed by the *DME Calculator* program.

Display Control The display control is used to display text generated by the user program. Additionaly to the label control, the font size can be changed to allow displaying of large numbers or text.

CheckBox Control The checkbox consists of a label with a small square in front where a checkmark can be placed or removed. It is commonly used to enable / disable certain functionality. The text of the label as well as the check mark can be modified or read out by the *DME Calculator* program.

Edit Control The edit control consists of a text input field and an ok button for accepting the input. This control allows the user to input text or numerical values.

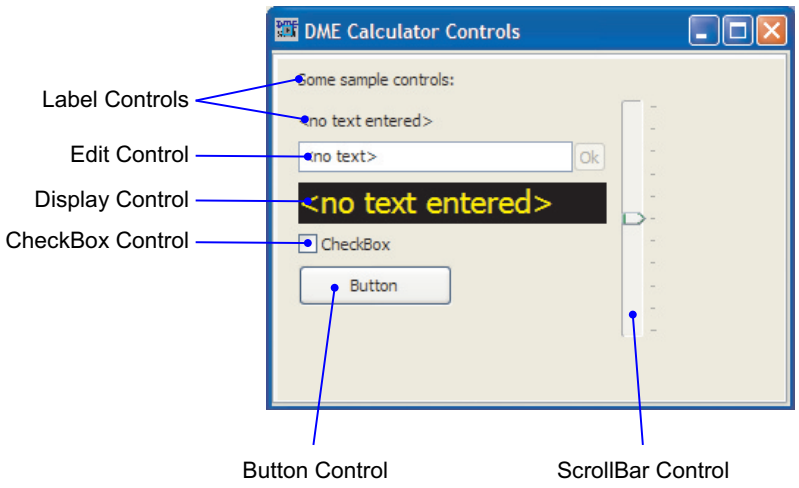


Figure 2.1: A GUI window with some sample controls.

ScrollBar Control The scroll bar control lets the user manipulate numerical values. This is especially useful if some parameters have a certain predefined range of allowed values.

The GUIs are commonly created with *DME Calculator*'s builtin GUI editor and then loaded in a *DME Calculator* program by the *loadgui()* function. This chapter describes how to create GUIs and use them by giving some examples.

2.2 Creating GUIs

To open the editor, choose *Create / Edit GUI...* from the file menu. The program asks for a file name. If a non existing file name is given, a new file is created, otherwise an existing file is opened an can be modified.

After the file name has been specified, the GUI editor window is opened (see Fig. 2.2). Controls are now added by pressing one of the buttons in the upper left corner and clicking somewhere in the workspace to place the control.

While editing, the controls are indicated as frames. Each control gets its own ID number which is later used to address the control from a *DME Calculator* program. By pressing the test button, the final appearance of GUI window can be investigated.

By double clicking inside a frame, the property dialog of the corresponding control is displayed. Here the appearance and other parameters can be set, depending on the type of control.

A single control can be selected by a single click inside a frame. To select more than one control, one can draw a selection rectangle by pressing and holding the left mouse button in a free area of the workspace and dragging the mouse. All controls in the

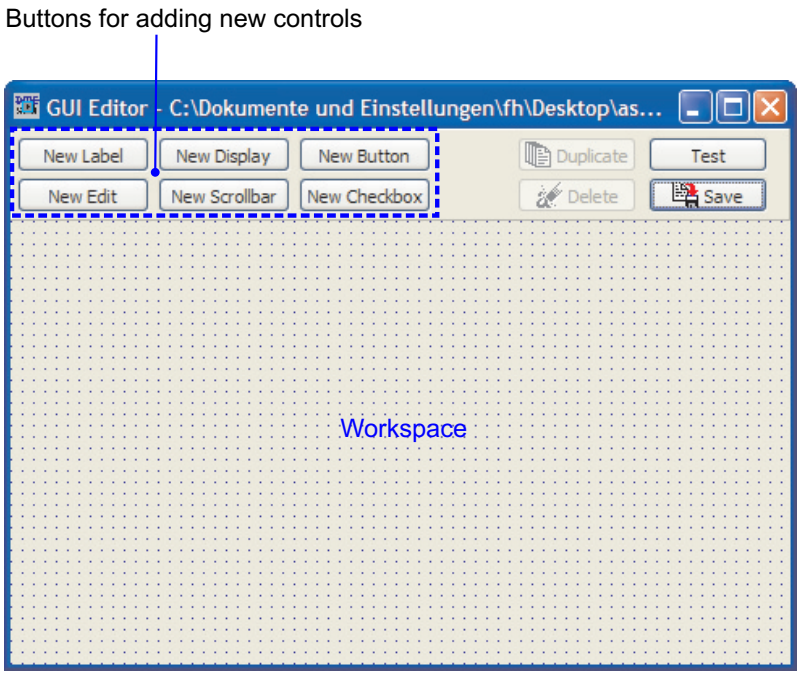


Figure 2.2: The GUI editor window.

rectangle become selected. The selected controls can either be duplicated or deleted by pressing the corresponding button.

The GUI file is saved to disk either by pressing the *Save* button or closing the editor window. The file format of the GUI file is quite simple: It is a comma separated text file (CSV) where each line describe a single control. Here comes a description of the single elements:

1. Control type id:
   0 = Label, 1 = Display, 2 = Scrollbar, 3 = Edit, 4 = Button, 5 = Checkbox

2. Left coordinate

3. Top coordinate

4. Right coordinate

5. Bottom coordinate

6. Control Id

7. A list of further parameters, depending on the control type

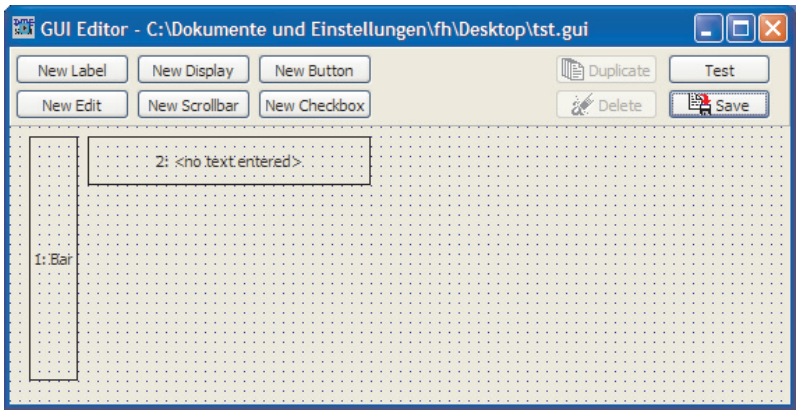


Figure 2.3: Sample GUI window

2.3 Using GUIs

For learning how to use GUIs we start with a small example. First, create a small GUI window named *tst.gui*. It should contain a scrollbar control with id 1 and a display control with id 2, as shown in Fig. 2.3. Close the editor window and from the *Edit* menu select *Insert GUI code...* and choose the previously generated editor file *tst.gui*. The following code is then created:

```
loadgui(0;'...\tst.gui'),
while(waitguievent(0);
 println('GUI Event...')
)
```

Example 2.3.1: Generated code example

This small piece of code displays the GUI and then waits for a user event. User events are generated for example by pressing a button, moving a scroll bar or editing text. In the code above, on each event the program displays a line „GUI Event...". To now do something else, we modify the code as follows:

```
loadgui(0;'...\tst.gui'),
setproperty(0;2;0;''),
while(waitguievent(0);
 a = getproperty(0;1;0),
 setproperty(0;2;0;a)
)
```

Example 2.3.2: Reading and setting properties

The code above now obtains the current value of the scroll bar and displays it in the display control.

2.3.1 Communication with GUI controls

For communication with the GUI controls, the functions *getproperty()* and *setproperty()* are used. There can be more than one GUI window loaded, the first parameter for both functions *getproperty()* and *setproperty()* is the window number. In the code above, we have specified to load the gui with the *loadgui()* function as window 0, so the first parameter in both functions tells to use the controls on window 0. The second parameter for both functions is the control id, which is displayed in the GUI editor when the test mode is not active. The third parameter for both functions is the property id. Each control type has different properties, for example the first property with id 0 of a scroll bar is its current value. The properties with id 1 and 2 are the minimum and maximum values of the scroll bar, corresponding to the values when the slider is moved fully up or down. The properties 3 and 4 are the page size and line size, corresponding to how much the scroll bar is moved when the user presses the page up and down keys or the up and down arrows.

For a full list of property ids for each control go to the online help of the functions *setproperty()*.

The code below now uses the parameter from the scroll bar to plot a curve, where the oscillation period depends on the scroll bar value.

```
loadgui(0;'...\tst.gui'),
setproperty(0;2;0;''),
while(waitguievent(0);
 a = getproperty(0;1;0),
 setproperty(0;2;0;a),
 plot(0;0;x;
  sin(x*(1+a/100))*exp(-x/(2*pi()));
 0;5*pi();200)
)
```

Example 2.3.3: Using GUI parameters

As a last step we will now add a button to reset the scrollbar to the middle position. We start with opening the GUI editor again and selecting the file *tst.gui*. Now we add a button, which will become control number 3. By double clicking on the button control frame, change the button text from „Button" to „Reset".

Close the GUI editor again and modify the code as follows:

```
loadgui(0;'...\tst.gui'),
setproperty(0;2;0;''),
while(waitguievent(0);
 if(haschanged(0;3);
  setproperty(0;1;0;50);
 0),
 a = getproperty(0;1;0),
 setproperty(0;2;0;a),
 plot(0;0;x;
  sin(x*(1+a/100))*exp(-x/(2*pi()));
  0;5*pi();200)
)
```

Example 2.3.4: Using GUI parameters

We have basically added 3 lines after the *while*: The *haschanged()* function checks, whether a control generated an event since the function *haschanged()* for the same control was called the last time. In this example this function is used to find out whether the button 3 has been pressed. If yes, the current value of the scroll bar is set to 50.

Chapter 3

Language reference

This chapter describes the language used in the *DME Calculator*. This language can perform calculation tasks, perform iterations and loops, and execute functions.

3.1 Syntax

The *DME Calculator*'s language is an *expression based* language, where *expression* means something that evaluates into a value. *Expression based* means that the whole program is an expression which is commonly made from simpler expressions also containing simpler expressions, etc. The simplest expression is just a number, like `42`. Another type of simple expression is a text string, like `'hello'`. Text strings are enclosed in single quotes. The third type of simple expression is a variable name, like `a` or `hugo`, which itself holds either a number or a text string.

A more complex expression includes an operator, e.g. a plus (+) sign. `1+1` is also an expression, it evaluates into the value `2`. Another example is `a+b`, which evaluates in to the contents of the variables a and b added together. Also possible is `'hel'+'lo'`, which evaluates into the text string `'hello'`. The + operator actually concatenates two expressions, it is also called an *infix*-operator. This is the only kind of operators the *DME Calculator* understands. The behaviour of the operator can be different, depending on whether it is used with in conjunction with strings or numbers, as shown on the + operator. Each infix-operator has an implicit *precedence* value. If one imagines the expression `1+4*5`, the multiplication must take place before the addition is executed, therefore the `*` operator has a higher *precedence* than the + operator. To change this default behaviour, the *DME Calculator* language knows round brackets `()`; with an expression like `(1+4)*5` one can force the evaluation of the addition to take place before the multiplication.

Another kind of expression is a function, like `cos(0)`. This example evaluates into the value 1. A function has one or more arguments, which are themselves expressions. It performs an action on its arguments and returns a string or a number. For example the

modulo function `mod` has two arguments. It calculates the remainder of the division of the left argument by the right argument. In the *DME Calculator* language, multiple arguments are separated by the semicolon (`;`). Therefore the expression `mod(8;3)` evaluates into `2`.

This is all we need to make a programming language. Everything else fits into this basic structure. Since a *DME Calculator* program is only an expression, all examples shown in this chapter can be directly entered into the program window. When the start button is pressed, the expression is evaluated and the result is displayed in the status line.

You can try this directly: Enter the expression `2*3` in the program window and press the start button or F2. The result (6) is displayed in the status bar.

3.2 Operators

The *DME Calculator* knows four different kinds of operators, which are mathematical operators, assignment operators, logical (also called 'boolean') operators and the special comma (`,`) operator. Mathematical operators are `*`, `+`, `/`, and − which are self explanatory and `^` which means exponentiate. The `+` operator has two different functions as when used together with numerical operands it performs an addition and with string operands it performs a concatenation. The other mathematical operators work only with numerical operands[1].

The second class of operators are assignment operators. The simplest assignment operator is the `=` operator. It is used to assign a value to a variable, for example `a=5` or `c='hello'`. As `a=5` is also an expression, it must have a return value. The result of such an expression is the value on the right side, which means `5` in this case. So the expression `(c=2)+5` is valid and has the return value `7` and at the same time the variable `c` gets `2`. The assignment operator has a lower priority than a mathematical operator and therefore `c=2+5` is the same as `c=(2+5)` which both have the return value `7` which is also assigned to the variable `c`. There are also assignment operators that perform an assignment and a mathematical operation. These are `+=`, `-=`, `/=` and `*=`. They first perform the mathematical operation and then assign the result to the variable on the left side. So if `a` has the value 2 then after `a+=5`, `a` has the value `7`. The return value of these type of operators is the result of the mathematical operation, so the value `7` in the example above. The `+=` operator also works with strings, so if `s` has the value 'hello' then after `s+=' you'`, `s` has the value 'hello you'.

The third class of operators consists of the *and* operator `&`, the *or* operator `|`, the *equal* operator `==`, the *not equal* operator `!=` as well as the compare operators `>`, `<`, `>=`, and

[1] If these operators are nevertheless used with string operands the strings are first translated into a numerical value. So the expression `'5'-'2'` is the same as `5-2`.

<=. These operators are used with numerical operands, where a zero means *false* and everything else means *true*. When a condition is true, the operators return the value `1` and a zero otherwise. So the expression `1==2` evaluates to `0` and `6>2+3` evaluates to `1`, whereas `1==2 & 6>2+3` evaluates to `0`. These logical operators are mainly used together with the `if()` or `while()` functions to control program flow[2].

Now there is one last operator that does not fit in the other classes, the comma (`,`) operator. This operator does nothing except return the value on the right side. This means, the expression `2,3` evaluates to `3`. This operator is commonly used for causing expressions to be evaluated. For example `a=2,b=3,c=55` assignes values to the variables a, b, and c. The return value of this expression is 55, as this is the right most expression. As the comma operator has the lowest precedence, one does not need to write the above example `(a=2),(b=3),(c=55)`, which is actually the same but requires more typing.

```
println('Hello'),
println('You')
```

Example 3.2.1: The comma operator can be regarded as a simple command delimiter.

One can also regard this operator as a 'command delimiter', simply executing expressions sequentially, as shown in Example 3.2.1.

3.3 Variables

Unlike other programming languages, in the *DME Calculator* language variables do not need to be declared. If a value is assigned to a variable and the variable did not exist before, it is automatically created. For example the expression `a=2` creates the variable a (if it did not exist before) and sets it to the value 2. If it is attempted to access a variable that does not exist, an error message is generated and the execution is stopped. For example when the expression `a=2+b` is executed and b does not exist, an error message is generated. The *DME Calculator* language knows two different kinds of variables: Scalar and array variables. A scalar variable can hold a numerical value[3] or a string.

An array can hold a theoretically unlimited number of numerical values, but no strings. An index is used to reference a specific value in the array. The index is given in square brackets. In the expression `a[5]=3` the fifth value of the array *a* is set to 5.

[2]There is no *not* operator, as the *DME Calculator* only knows infix operators with two arguments. For this purpose please use the `not()` function.

[3]Numerical values in the *DME Calculator* are always floating point numbers having *double* precision. They can hold values in the range $-1.7 \cdot 10^{308} \ldots 1.7 \cdot 10^{308}$. The smallest value not unequal to zero is $1.7 \cdot 10^{-308}$.

Variable names in the *DME Calculator* are case sensitive, so the variable *a* is different from the variable *A*. Arrays are commonly used for processing multiple values in a loop

```
a[0]=3,
a[1]=5,
a[2]=2,
sum=0,
for( i=0; i<length(a[]); i+=1;
     sum += a[i]),
sum2=0,
foreach( i; a[]; sum2 += a[i])
```

Example 3.3.1: In this example the array *a* is initialized with 3 values. Then these values are summed in a for-loop and in a for-each-loop. After execution both *sum* and *sum*2 have the value 10.

like shown in Example 3.3.1. The first value in an array always has the index 0. The length of the array is determined by the highest index that has been assigned. In Example 3.3.1 the highest index that has been assigned is 2, therefore the array has the length 3, i.e. it contains 3 values.

```
a[99]=5,
a[27]
```

Example 3.3.2: This program returns zero, as a[27] is automatically initialized with zero.

When assigning a value to an index which is higher than the original length of the array, the array length is automatically increased and all values between the old length of the array and the newly assigned value are initialized with zeros. If the program in example 3.3.2 is executed, the array *a* immediately has a length of 100 after the first assignment, and all values below 99 are initialized with zeros. So the return value of the whole program is zero, as `a[27]` is zero.

3.3.1 Global Variables

There also exists a special kind of variables, called 'global variables'. Unlike other variables, these are valid not only in the expression where they are defined but also within user defined functions. They also can be initialized in the global panel and are then accessable within a normal calculator panel. Global variables are defined by a leading $ sign in front of their names.

```
$myglobal = 5,
define(f(x);x * $myglobal),
f(3)
```

Example 3.3.3: Global variables

In this example a global variable `$myglobal` is defined. This is then used within a user defined function. The example returns the value 15.

3.4 Comments

```
a = 2, /* this is a comment */
b = 3, /* this is also a comment
that extends over two lines */
c = /* another comment */ 'hello'
/* the last line assigns 'hello' to c */
```

Example 3.4.1: Using comments

It is often necessary to write some descriptive text into the program. To mark the beginning of descriptive text, use `/*`. Then the parser ignores everything until a `*/` is encountered. See Example 3.4.1.

3.5 Functions

This section describes some basic functions contained in the *DME Calculator* language. For the description of all other functions have a look at the online help of the *DME Calculator*. Some functions require a whole array as an argument, like the `arraymax()` function. To pass a whole array as an argument, use the array name with the square brackets and without an index, so the expression `arraymax(a[])` returns the biggest value in the array. In the *DME Calculator* language function names are case insensitive.

3.5.1 Program flow functions

The following functions are used for controlling the program flow, i.e. for creating loops and conditional evaluation. These functions also have a return value which is often not used.

for(a;b;c;d)

This function is commonly used to create a loop that is based on a counter variable. When this function is evaluated, first the expression *a* is evaluated. Commonly here a counter variable is initialized. The second expression *b* is a condition. If this returns true (1), expression *d* is evaluated, which is the "working function" of the loop. After this, expression *c* is evaluated, which is commonly used for incrementing a counter variable. Then the procedure repeats with evaluating expression b again. The return value of the `for()` is the result of the last evaluation of expression *d* or 0 if expression *d* is has not been evaluated.

```
for(i=0;i<10;i+=1;
 println(i)
)
```

Example 3.5.1: The examples prints the numbers 0 to 9 to the text output window.

foreach(i;a[];c)

This function is a specialized *for* function for processing arrays. *i* must be a variable, which is used as an array iterator. *i* counts from zero to the length of `a[]`-1 while for every count expression `c` is evaluated. See Example 3.3.1 on page 24.

if(a;b;c)

This function first evaluates the expression `a`. If this is true (meaning it evaluates to 1), expression `b` is evaluated, otherwise expression `c` is evaluated. The return value of the `if` is the return value of either expression `b` or expression `c`, depending on which one has been evaluated. Example 3.5.2 shows two usage examples, one uses the return value of

```
/* first possibility */
r = if(a;'yes';'no'),
/* second possibility */
if(a;r='yes';r='no')
```

Example 3.5.2: Two different possibilities using the if() function.

the `if` function and the other sets a variable inside the `if`.

sleep(x)

Waits for *x* milliseconds, so `sleep(1000)` holds progam execution for 1 second. The return value is always 1.

while(a;b)

This function executes expression `b` as long as expression `a` evaluates to true (1). `a` is evaluated before `b`. Example 3.5.3 shows an infinite loop permanently writing 'hello'+linefeed

```
s='baud=9600 parity=N stop=1 data=8',
commopen(0;s;3000),
while(1;
 commwrite(0;'hello'+chr(13))
)
```

Example 3.5.3: Here the while() is used to create an infinite loop.

to the 1st serial port.

3.5.2 User interface functions

The following functions are used together with *DME Calculator*'s graphical user interfaces.

getproperty(panel;control;id)

This function returns the value of a control property. The control is addressed by a panel number and a control number. The value of the property determined by *id* is returned. The control number is the number indicated in the gui editor. The panel number is the number which has been passed to the *loadgui()* function when the GUI has been loaded.

It depends on the control type, which property ids are valid. Here comes a list of all controls with their properties:

Control Type	Property ID	Description
Label	0	Caption
	1	Alignment: 0 = left, 1 = middle, 3 = right
Display	0	Caption
	1	Alignment: 0 = left, 1 = middle, 3 = right
ScrollBar	0	Current Value
	1	Minimum
	2	Maximum
	3	Page Size
	4	Line Size
Edit Object	0	Caption
	1	Alignment: 0 = left, 1 = middle, 3 = right
Button	0	Caption
	1	Currently pressed (0 = not pressed)
CheckBox	0	Caption
	1	Checked or unchecked (0 = unchecked)

haschanged(panel;control)

This function returns a value unequal to 0 when a control has generated an event since the last time this function has been called for the same control. The control is addressed by a panel number *panel* and the control number *control*. In example 3.5.4, controls 1 and

```
loadgui(0;'...\someexample.gui'),
while(waitguievent(0);
 if(haschanged(0;1);
  println('Button 1 has been pressed')
 0),
 if(haschanged(0;3);
  println('Button 3 has been pressed')
 0)
)
```

Example 3.5.4: An example for *haschanged*

3 are checked whether there has been an event generated from these and a corresponding text is written to the display.

loadgui(panel;filename)

The function *loadgui* loads the GUI file *filename* as panel number *panel* and displays it. There can be more than one GUI loaded at the same time, so that all GUI functions need the panel number as a first parameter to address the right panel. The *loadgui* function is the first function to use before any other of the GUI functions.

panelname(panel;txt)

This sets the header text for a panel for the case that more than one GUI file is loaded at the same time. If only one GUI file is loaded, no header text is displayed.

setproperty(panel;control;id;value)

This sets the property *id* of control *control* on panel *panel* to *value*. For a list of possible property IDs see function *getproperty* on page 27.

waitguievent(timeout)

This stops program execution until the user generates a user interface event on a panel. Events are created for example by pressing a button, moving a scroll bar or changing the text in an edit control. If *timeout* is not zero, this function returns as well when *timeout* milliseconds have elapsed without the user generating an event.

The function returns 0 when the timeout was reached and a value unequal to 0 when the user caused an event. If the function returns 0, for example *haschanged()* can be used for checking which control generated the event.