Macro Language Syntax

Specifies the NIS-Elements AR Macro Language features.

Variable types

The following data types are implemented:

char text

char8 <-128, +127>

byte <0, +255>

int <-32.768, +32.767>

int64 <-9.223.372.036.854.775.808, +9.223.372.036.854.775.807>

word <0, +65.535>

long <-2.147.483.648, +2.147.483.647>

dword <0, +4.294.967.295>

double <1.7E +/- 308 (15 digits)>

Structures and Unions

Structures and unions are not supported.

Arrays

One and two dimensional arrays are supported.

Local and Global Variables

You should declare local variables at the beginning of macro or function only. You should declare global variables only at the beginning of macro. You can run two nested macros declaring the same global variables, but they must be of the same type. You can declare global variables by prefixing the declaration *global*. Eg.:

```
global int Number_Rows;
global char buffer[200];
```

The “global” keyword in front of the variable definition assigns the variable to the global scope. Such variable is then accessible from all function scopes within the macro interpreter.

Statements

Supported Statements

for The *for* statement lets you repeat a statement a specified number of times.

```
Syntax

for([init-expr]; [cond-expr]; [loop-expr])
         statement
```

First, the initialization (init-expr) is evaluated. Then, while the conditional expression (cond-expr) evaluates to a non zero value, Statement is executed and the loop expression (loop-expr) is evaluated. When cond-expr becomes 0, control passes to the statement the following the for loop.

while The *while* statement lets you repeat a statement until a specified expression becomes false.

```
Syntax

while(expression) statement
```

First, the expression is evaluated. If expression is initially false, the body of the while statement is never executed, and control passes from the while statement to the next statement in the program. If expression is true (nonzero), the body of the statement is executed and the process is repeated.

if, else Conditionally executes a statement or group of statements, depending on the value of an expression.

```
Syntax
if(expression) statement1
     [else statement2]
```

The *if* keyword executes statement1 if expression is true (nonzero); if *else* is present and expression is false (zero), it executes statement2. After executing statement1 or statement2, control passes to the next statement.

goto Transfers control of the program execution.

```
Syntax

goto name;
      . . .
    name: statement
```

You cannot use goto to jump inside the block from outside. E.g. following is not allowed:

```
goto label;
    if(k>5)
      {
    label:
       DilateBinary(3, 5);
       FillHoles();
       }
```

You cannot use goto to jump out from the block more then 2 block levels down. E.g. the following is not allowed:

```
for(i=0; i<64; i=i+1)
       {
       for(j=0; j<64; j=j+1)
          {
          if(a[i] > b[i])
             {
             value = i;
             goto end;  // crossing 3 right brackets
             }
           }
        }
    end
```

break Terminates the execution of the nearest enclosing statement.

```
Syntax

break;
```

The *break* keyword terminates the execution of the smallest enclosing *for* or *while* statement in which it appears. Control passes to the statement that follows the terminated statement.

continue Passes control to the next iteration of the statement in which it appears.

```
Syntax
continue;
```

The *continue* keyword passes control to the next iteration of the *for* or *while* statement in which it appears. Within a while statement, the next iteration starts by reevaluating the expression of the while statement. Within a for statement, the first expression of the for statement is evaluated. Then the compiler reevaluates the conditional expression and, depending on the result, either terminates or iterates the statement body.

Statements which are not supported

do, switch, case, default, typedef These statements are not supported.

Directives

The following directives are supported by the system:

define The *#define* directive assigns a meaningful name to a constant in a program.

```
//Syntax

    #define identifier token-string
```

The directive substitutes token-string for all subsequent occurrences of the identifier in the source file. Token-string can be a value or a string (only for 32-bit version of NIS-Elements AR).

```
//Example

    #define ERROR_SPRINTF 0
    #define MAINDIR "c:\Images"

    int main()
    {
    char buf[256];
    int retval;

    retval = sprintf(buf, "%s", "MAINDIR");
    if(retval == ERROR_SPRINTF)
    Beep();
    else WaitText(0., buf);
    return TRUE;
    }
```

include Specifies the name of the file to be included.

```
//Syntax

    #include filename
```

The *#include* directive includes the contents a file with a specified filename in the source program at the point where the directive appears.

```
Example
    // if you do  not specify the full path, NIS-Elements
assumes, that it is a relative path to a main directory

    #include "macros\my_macro.h"
    #include "c:\NIS-Elements\macros\my_macro1.h"
```

import The *#import* directive is used to incorporate information from an external library.

```
//Syntax

    #import("DLLname");
    #import function_declaration
```

NIS-Elements can call functions from external DLL's. You should import the DLL, where the functions resides and then make a declaration of the functions. You should not import the

following system DLL's: kernel32.dll, user32.dll, gdi32.dll, com32.dll, comdlg32.dll. This feature is available only for 32 bit version of NIS-Elements.

```
//Example

    #import("luc_13.dll");
    #import int RTF_ReplaceVariables(LPSTR destfile, LPSTR
sourfile);
    #import int RTF_FindQuestion(LPSTR sourfile, LPSTR question,
long *length, LPSTR defvar);
    #import int RTF_ReplaceQuestion(LPSTR destfile, LPSTR
sourfile, LPSTR replacement);
```

__underC The *__underC* directive assigns a function to be interpreted by the UnderC engine instead of the standard interpreter.

```
//Syntax
//works only on 32bit operating systems

__underC int inter_sharpen(int cols, int rows) { }
```

#importUC The *#importUC* directive imports an API function to the UnderC engine so that it can be used from within there.

```
//Syntax
//works only on 32bit operating systems

#importUC DisplayCurrentPicture;
```

Example 9. Usage of #importUC / __underC

```
#importUC DisplayCurrentPicture;

int main()
   {
   int rows, cols;

   AddUndoImage();

   rows = 256;
   cols = 256;

   SetCommandText("Working...");
   Get_Size(SIZE_PICTURE,NULL,NULL,&cols,&rows);

   //Enable_Interrupt_InScript(0);
   Enable_Interrupt_InScript(2);

   inter_sharpen(cols, rows);
```

```
    DisplayCurrentPicture();
    }

__underC int inter_sharpen(int cols, int rows)
    {
    int i, j, value;
    for(i=0; i<rows; ++i)
   {
   for(j=0; j<cols; ++j)
      {
      value = 5 * GetPixelValue(-1,i,j,0);
      value = value - GetPixelValue(-1, i-1,j, 0);
      value = value - GetPixelValue(-1, i+1,j, 0);
      value = value - GetPixelValue(-1, i,j+1, 0);
      value = value - GetPixelValue(-1, i,j-1, 0);
      if(value <0) value=0;
      if(value >255) value=255;
      SetPixelValue(0,value,i,j,0);
      }
   if(0 == i % 20)
      DisplayCurrentPicture();
   }
    }
```

Operators

The following operators are supported by the system. If an expression contains more than one operator, the order of operations is given by the priority of operators. Following operators have higher priority then the others: / * %. You should use brackets to define evaluation order other than implemented in NIS-Elements, which is: from right to the left.

arithmetic operators

+ Addition

- Subtraction

***** Multiplication

/ Division

assignment operators

The assignment operator assigns the value of the right operand to the left operand.

= Addition

bitwise operators

The bitwise operators compare each bit of the first operand to the corresponding bit of the second operand. The following bitwise operators are supported.

& Bitwise AND

| Bitwise OR

~ One's complement

pointer operators

& Address of ?

***** Indirection

relational operators

< Less than

<= Less than or equal to

> Greater than

>= Greater than or equal

== Equal

!= Not equal

logical operators

The logical operators perform logical operations on expressions. The following logical operators are supported:

&& Logical AND

|| Logical OR

! Logical NOT

Expression evaluator

Expression evaluator supports the precedence of operators / * %. It evaluates the expression strictly from right to left, so you need to use brackets.

C-like functions

The system can interpret your own C-like functions. An entry point to program is the main() function in your macro. If main() is not presented, for backward compatibility, the body of the macro is also considered as "main function". The new macros should use main() as an entry point.

The general C-like functions (also called interpreted functions, as opposed to basic system functions from NIS-Elements) have the following syntax.

Syntax

```
int MyFunction(int a, LPSTR str, double d)
{
int retval;
. . .
return retval;
}
```

Return Value The return value can be any basic data types (char, int, word, dword, int, double or pointer).

Parameters Parameters can be any basic data types (char int, word, dword, int, double or pointer).

> **Note**
>
> Int and double parameters are automatically converted to string type, when they are assigned to text variable, or are one of the parameters of a function.

Example

```
int main()
{
char buf[256];
my_function1(buf);
WaitText(0., buf);
return TRUE;
}

int my_function1(char *buf)
{
strcpy(buf, "This function has a pointer to char array as a
parameter");
return TRUE;
}
```