RAM corruption reasons in mP/mB

2015-07-10

1 Contents

2 Purpose of this document

Make people aware about the ways of working that can introduce ram corruption (and of course how to avoid it). In this document the

many ways where it goes right are not mentioned, only the danger areas.

Some test code of examples of things going bad can be found in sections mP Test/Example code and mB Test/Example Code.

3 How to recognize Ram Corruption?

Ram corruption can cause several different phenomena:

- Unexpected changes of ram content (of course)
- Unexpected and unexplainable resets of the MCU
- Code that ceases to work (or starts again to work) when code is added/deleted or changed in a completely other (not related) code part. *This is the most odd phenomenon of ram corruption*.
- ...

The phenomena can occur separately or in combination.

4 Possible causes of Ram Corruption

Ram corruption has mostly only one reason: not respecting the boundaries of a variable, so writing to ram that does not belong to the variable:

- Exceeding array or string boundaries (see section 4.1)
- Not terminating a string with zero (see section 4.2)
- The result of the “SizeOf” function (see section 4.3)
- Using the ‘+’ operator for string concatenation (see section 4.4)
- Passing wrong variable types to var parameters (see section 4.5)

4.1 Exceeding array or string boundaries

Some ways to get into ram corruption by exceeding boundaries:

Example in mP:
```
var MyArr: array[10] of byte;
    I: byte;
  ...
  for I := 0 to 10 do  // the array boundaries go from 0 to 9, not from 0 to 10!
                       // One byte of another variable will be overwritten!
  MyArr[I] := I;
```

Example in mB:

```
dim MyArr as byte[10]
    I as byte
  ...
  for I = 0 to 10   ' the array boundaries go from 0 to 9, not from 0 to 10!
                    ' One byte of another variable will be overwritten!
  MyArr[I] = I
  next I
```

<u>Example using “SizeOf” in mP:</u>

```
var MyArr: array[10] of byte;
    I: byte;
  ...
  for I := 0 to SizeOf(MyArr) do // the array boundaries go from 0 to
                                 // SizeOf(MyArr) -1 , not from 0 to SizeOf(MyArr)!
                               // One byte of another variable will be overwritten!
  MyArr[I] := I;
```

<u>Example using “SizeOf” in mB:</u>

```
dim MyArr as byte[10]
    I as byte
  ...
  for I = 0 to SizeOf(MyArr)  ' the array boundaries go from 0 to
                              ' SizeOf(MyArr) -1 , not from 0 to SizeOf(MyArr)!
                              ' One byte of another variable will be overwritten!
    MyArr[I] = I
  next I
```

<u>Example in mP:</u>

```
var MyStr: string[5];
  ...
  MyStr := ‘abc’;
  StrCat(MyStr, ‘def’); // string will become length 6.
                        // One byte of another variable will be overwritten!
```

<u>Example in mB:</u>

```
dim MyStr as string[5]
  ...
  MyStr = ‘abc’
  StrCat(MyStr, ‘def’) ' string will become length 6.
                       ' One byte of another variable will be overwritten!
```

<u>Example in mP:</u>

```
var MyStr: string[5];
    MyWord: word;
  ...
  MyWord := 300;
  IntToStr(MyWord, MyStr); // IntToStr needs a string of string[6] as its
                           // destination.
```

```
                                  // One byte of another variable will be overwritten!
```

Example in mB:

```
dim MyStr as string[5]
    MyWord as word
  ...
  MyWord = 300
  IntToStr(MyWord, MyStr)  ' IntToStr needs a string of string[6] as its
                            ' destination.
                            ' One byte of another variable will be overwritten!
```

4.2 Not terminating a string with zero

A string should always have a terminating zero byte. Normally the routines made to handle strings (as the ones in the String library) take care of that.
But: when manipulating a string by byte banging (if I may use that term), the terminating zero must also be not forgotten.

Important: all **string parameters** of **any string function** (e.g. from the String library) must be **correctly zero terminated**!
Using a string function (e.g. StrLen, StrCat, etc...) with unterminated strings as their parameters will give very unpredictable results and may result in ram corruption!

4.2.1 The StrnCpy function

The **StrnCpy** function specification is such that one can **not always** expect the resulting string to be terminated by a zero. In fact, there is no 'terminating' zero in the result, only padding with zeroes is done (only when the resulting "string" to too short, < n). In janni's replacement library the StrnCpy function always adds a terminating zero.

This specification can give some strange results, see http://www.mikroe.com/forum/viewtopic.php?f=81&t=64329&p=256785&hilit=strncpy#p256785

4.2.2 Concatenate a variable string with a constant string

This can result in a not terminated string, see section 4.4.1.

4.2.3 Other

Example in mP:

```
var MyStr: string[5];
    SomeArray: array[10] of byte;
  ...
  memcpy(@MyStr, @SomeArray, 3); // If MyStr[3] was not zero before the memcpy
                                 // then MyStr has no terminating zero.
                                 // Following operations based on e.g. Str_Len(MyStr)
                                 // will go horribly wrong...
```

Example in mB:

```
dim MyStr as string[5]
    SomeArray as byte[10]
  ...
  memcpy(@MyStr, @SomeArray, 3) ' If MyStr[3] was not zero before the memcpy
                                ' then MyStr has no terminating zero.
                                ' Following operations based on e.g. Str_Len(MyStr)
                                ' will go horribly wrong...
```

The correct way of working in this case is:

Example in mP:

```
var MyStr: string[5];
    SomeArray: array[10] of byte;
  ...
  memcpy(@MyStr, @SomeArray, 3);  // copy 3 characters and
  MyStr[3] := 0;                  // add terminating zero.
```

Example in mB:

```
dim MyStr as string[5]
    SomeArray as byte[10]
  ...
  memcpy(@MyStr, @SomeArray, 3) ' copy 3 characters and
  MyStr[3] = 0                  ' add terminating zero.
```

4.3 The "SizeOf" usage

Normally the 'SizeOf" function is very handy when e.g. copying or setting all bytes of a variable.

Example in mP:
```
var arr1, arr1: array[10] of char;
...
  memcpy(arr1, arr2, SizeOf(arr1)); // instead of
  memcpy(arr1, arr2, 10);
```

Example in mB:
```
dim arr1, arr1 as char[10]
  ...
  memcpy(arr1, arr2, SizeOf(arr1)) ' instead of
  memcpy(arr1, arr2, 10)
```

SizeOf gives the size of a variable (or a type, or a constant) in bytes.

But: there are some pitfalls:

- using SizeOf on "var" parameters ("byref" parameters in mB), section 4.3.1
- using SizeOf on "external" string or array items, section 4.3.2
- using SizeOf on pointers, section 4.3.3.

4.3.1 Using SizeOf on Var (ByRef in mB) parameters

Complex parameters are always to be passed to subroutines as "var" parameters, this is a constraint of mP. This means that only a reference of the variable is passed to the subroutine.

Different sizes of formal and actual parameters are allowed in mP and mB for flexibility reasons, otherwise separate subroutines should be created for e.g. every size of a string parameter or array parameter.

Example in mP:
```
var S1: string[10];
...
procedure Process_String(var Str: string[20]); // formal parameter is "string[20]"
begin
end;
...
  Process_String(S1); // actual parameter is "string[10]"
```

Example in mB:

```
dim S1 as string[10]
  ...
sub procedure Process_String(dim byref Str as string[20]) ' formal parameter is
"string[20]"
'begin sub
end sub
...
  Process_String(S1) ' actual parameter is "string[10]"
```

When using the SizeOf function on a var parameters ("byref" parameters in mB), the size returned is always that of the one defined in the formal parameter (the one in the subroutine definition), not the actual parameter (the one passed to the subroutine during a call).

So, using the SizeOf on parameters has to be done carefully, because of possible difference between formal and actual parameter size.

Example in mP:

```
var S1: string[10];
...
procedure Process_String(var Str: string[20]); // formal parameter is "string[20]"
begin
  memset(@Str, 0, SizeOf(Str)); // 21 bytes are cleared!
end;
  ...
  Process_String(S1); // 21 bytes in stead of 10 bytes are cleared: outside string
```

Example in mB:

```
dim S1 as string[10]
  ...
sub procedure Process_String(dim byref Str as string[20]) ' formal parameter is
"string[20]"
'begin sub
  memset(@Str, 0, SizeOf(Str)) ' 21 bytes are cleared!
end sub
  ...
  Process_String(S1) ' 21 bytes in stead of 10 bytes are cleared: outside string
```

Above example will surely lead to data corruption, because the routine assumes the parameter Str being "string[20]" while the actual parameter is "string[10]". This means that 21 bytes will be filled with zero -- SizeOf(string[20]) is 21—in stead of 11.

In some other circumstances, where the e.g. actual parameter has more bytes than the formal one, not all bytes will be cleared...

The solution in this case is also making the actual size of the parameter an extra parameter of the routine,

Example in mP:
```
var S1: string[10];
...
procedure Process_String(var Str: string[20], Size: word);
begin
  memset(@Str, 0, Size);
end;
...
  Process_String(S1, SizeOf(S1));
```

Example in mB:
```
dim S1 as string[10]
  ...
sub procedure Process_String(dim byref Str as string, Size: word[20])
'begin sub
  memset(@Str, 0, Size)
end sub
...
  Process_String(S1, SizeOf(S1))
```

Make sure that "SizeOf(S1)" has the correct value i.e. it is not a var or byref routine parameter!

In the case the "var" parameter is a string, then the size of it can be left out in the formal routine definition,

Example in mP:
```
procedure Process_String(var Str: string); // formal parameter is "string"
begin
end;
```

Example in mB:
```
sub procedure Process_String(dim byref Str as string) ' formal parameter is "string"
'begin sub
end sub
```

But: in this case "SizeOf(Str)" will give zero: the compiler does not know its (formal) size! This can be very dangerous e.g. when the compiler has to do string concatenation with the '+' sign, see section Using the '+' operator for concatenation inside functions.

4.3.2 Using SizeOf on external string or array items

The using unit has to provide some “dummy” size if the actual size is not known to that using unit. Of course it is better when the size of the “external” definition in the using unit is the same as the actual one in the used (external) unit. See sections mP Test/Example code and mB Test/Example Code for an example.

However, the size of string of array items (variables or constants) declared as external in a unit (using unit) is correct provided the compiler can find the actual definition of the item:

- The using unit is the main unit and the used unit is present in the Project Manager, **or**
- The using unit is not the main one and the used unit is in the “uses” clause of the using unit.

If the compiler can not find the used unit (where the actual definition of the items is done), then it assumes the size to be the dummy one...

4.3.3 Using SizeOf on pointers

Using “SizeOf” on pointers (routine parameters or otherwise) is rather useless: the size returned will always be that of the pointer itself, never of the variable it points to.

4.4 Using the ‘+’ operator for string concatenation

4.4.1 Concatenate a variable string with a constant string

Here an error in present in the compiler, causing a string not to be terminated:

Example in mP:
```
var text: string[50];
    Str1: string[15];
    Siz : word;
const ConstString = 'ghi';
…
  Str1 := 'abc';
  text := Str1 + ConstString; // text = 'abcghighi'         <--ERROR (expected = 'abcghi')
  Siz := strlen(text); // expected strlen = 6; actual = 9  <--ERROR
```

Example in mB:
```
dim
  text as string[50]
  Str1 as string[15]
  Siz as word
const
  ConstString = "ghi"
```

```
...
Str1 = "abc"
text = Str1 + ConstString ' text = "abcghighi"         <--ERROR (expected = "abcghi")
Siz = strlen(text) ' expected strlen = 6; actual = 9   <--ERROR
```

4.4.2 Inside functions on function parameters

This is a very dangerous one. The compiler has to create a temporary string to hold the concatenated string. The size of this temporary string is calculated out of the sizes of the strings to be concatenated. The latter is the problem: If concatenation length is to calculated out of formal parameter sizes things can go wrong horribly.

Example1 in mP:

```
procedure Proc10(var Par: string[5]);
begin
  Par := Par + 'defghij'; // the tempstring created for the concatenation is only a
string[5] type !!!
end;
```

Example1 in mB:

```
sub procedure Proc10(dim byref Par as string[5])
'begin sub
  Par = Par + "defghij" ' the tempstring created for the concatenation is only a
string[5] type !!!
end sub
```

Example2 in mP:

```
procedure Proc11(var Par: string);
begin
  Par := Par + 'ghi'; // the tempstring created for the concatenation is only a string[2]
type !!!
end;
```

Example2 in mB:

```
sub procedure Proc11(dim byref Par as string)
'begin sub
  Par = Par + "ghi" ' the tempstring created for the concatenation is only a string[2]
type !!!
end sub
```

As you can see, in both examples above, the temporary string created to hold the concatenated one is much too short!

The solution here is never to use the '+' operator to concatenate strings, but e.g. use StrCat. In this case no temporary string needs to be created by the compiler.

Example in mP:
```
procedure Proc10(var Par: string[5]);
begin
  StrCat(Par, 'defghij');
end;
```

Example in mB:
```
sub procedure Proc10(dim byref Par as string[5])
'begin sub
  StrCat(Par, "defghij")
end sub
```

but it is still the responsibility of the code to stay within the string boundaries, see section Exceeding array boundaries.

4.4.3 When passing string type parameters to functions

This is also a very dangerous one. The compiler has to create again a temporary string to hold the concatenated string before passing it on to the function. The size of this temporary string is the same as the function's string parameter "size". The latter is the problem: If the function's string parameter is defined too small then surely a ram corruption will occur.

In the example below the compiler has to create also temporary variables:

Example in mP:
```
var
    ADCZeroCounts: Integer;
    Res: word;
    S1: string[128];
...
procedure ShowStr(var Strng: string[10]);
begin
  UART1_Write_Text(Strng);
end;
...
WordToStr(ADCZeroCounts, s1);
ShowStr('Text1 ' + s1 + ' Text2' + #13 + #10);
```

Example in mB:
```
dim
  ADCZeroCounts as integer
  Res as word
  S1 as string[128]
  ...
sub procedure ShowStr(dim byref Strng as string[10])
'begin sub
  UART1_Write_Text(Strng)
end sub
...
WordToStr(ADCZeroCounts, s1)
ShowStr("Text1 " + s1 + " Text2" + chr(13) + chr(10))
```

As you can see the string parameter of the function ShowStr is defined as "String[10]". This means also that the temporary stringvariable created by the statement "`ShowStr("Text1: " + s1 + " Text2" + chr(13) + chr(10))`" will only be able to hold 10 characters, while the actual string passed to the function has 19 characters.

The solution is changing the function parameter definition:

mP:

```
procedure ShowStr(var Strng: string); // no size definition of the string here
```

mB:

```
sub procedure ShowStr(dim byref Strng as string) 'no size definition of the string here
```

Surprisingly, when the size of the string parameter is not defined (so "String" is used as the type), the compiler calculates the necessary length of the temporary string variable out of the statements wherein the concatination is done (here the call to "ShowStr"). So, in this case the size of the temporary variable will be 142 characters, which is more then enough of course.

Of course other solutions are also possible: e.g. do not use the '+' operator to concatenate strings, (e.g. create your own variable and use StrCat) but it is still the responsibility of the code to stay within the string boundaries, see section Exceeding array boundaries.

4.5 Passing wrong variable types to var parameters

Normally the type of formal (in the procedure or function definition) and the actual (during the procedure or function call) var parameters should be the same.
E.g. if a formal parameter is of type "word" then the actual parameter should also be.

In the example below the actual and formal var parameter types are the same: no Risk.

<u>Example in mP</u>

```
var w_: word;
procedure TestVarPar(var Wrd: word); // <-- here the formal parameter is defined
begin
   ...
   Wrd := some_result;
end;
...
  TestVarPar(w_); // <-- here the procedure/function is used with the actual parameter
```

<u>Example in mB</u>

```
dim w_ as word
sub procedure TestVarPar(dim byref Wrd as word) ' <-- here the formal parameter is
                                                ' defined
   ...
   Wrd = some_result
end sub
...
  TestVarPar(w_) ' <-- here the procedure/function is used with the actual parameter
```

Known allowed deviations from this rule are <u>arrays and strings</u>. It is impossible to create separate functions for every possible size of an array or string, so a size mismatch here is commonly used. See however the SizeOf usage (section 4.3) for the risks of doing that.

The current versions of mP and mB however do, besides allowing the deviations for arrays and strings, <u>implicit var parameter conversion</u>. This means that actual and formal parameters do not have to be the same type or the same size. When this conversion happens a warning is given however: “Suspicious pointer conversion”.

This implicit conversion gives a lot of freedom the mP/mB user, but also a lot of responsibility:
if the <u>actual and the formal parameters differ in size</u>, then a <u>possibility of ram corruption</u> arises. This is the case when 2 conditions are met:

- If the actual size is smaller than the formal size, and
- the var parameter is assigned a value inside the procedure.

<u>Example in mP</u>

```
var Bte_: byte; // <--- byte type
...
procedure TestVarPar(Var V1: word); // <-- formal parameter is of type word
begin
  V1 := $1234; // <-- the parameter is assigned a value inside the procedure
end;
...
  TestVarPar(Bte_); // <-- the actual parameter (byte) has a smaller size than the
                    // formal one, which is of type word
                    // This will cause RAM CORRUPTION
```

<u>Example in mB</u>

```
dim  Bte_ as byte ' <--- byte type
  ...
sub procedure TestVarPar (dim byref V1 as word) ‘ <-- formal parameter is of type word
  V1 = $1234 ' <-- the parameter is assigned a value inside the procedure
end sub
...
  TestVarPar(Bte_) ' <-- the actual parameter (byte) has a smaller size than the
                   ' formal one, which is of type word
                   ‘ This will cause RAM CORRUPTION
```

In above example both conditions are met.
The call to "TestVarPar" will cause ram corruption. The procedure assumes the parameter is a word (2 bytes), and assigning a value to is will fill 2 bytes in ram, not one (for a byte variable) as expected, the byte next in ram above "Bte_" will be overwritten with (in the example) the value $12. As expected the variable "Bte_" will get the value $34.

5 Tips

- If one can identify the corrupted variable (e.g. a string variable that has a corrupted text), it is fairly reasonable to assume that the source of the corruption is the variable placed right before the corrupted one. Just open the Statistics window (CTRL+ALT+S) and check the distribution of variables across memory (thanks **aCkO** for the tip).

- In the help of the Fat32 library for mP for PIC32 (and perhaps also for dsPIC and PIC) there is a **ram corruption** problem: the buffer for reading a number of bytes from a file is not declared, only the pointer to it.
  This results in card data to be written to the place where the Readbuffer "pointer" is located (which place is only a few bytes in size)... See
  http://www.mikroe.com/forum/viewtopic.php?f=172&t=64695&p=258170&hilit=corruption+pic32#p258170

6 mP Test/Example code

The main unit in mP:

```
program SizeOftest;

{ Declarations section }

type TArray  = array[10] of byte;
     TString = string[10];
     PArray  = ^TArray;
     PString = ^TString;

var Siz    : word;
    Strng : string[4];
    String10: TString;
    ChArr : array[15] of char;
    Arr    : Array[25] of byte;

    ExtString: string[1]; external;        // dummy size
    ExtArry  : array[1] of byte; external; // dummy size

    I: byte;

Const Carr: array[1] of byte; external;

function proc1(var Par: Tstring): word;
begin
  Result := SizeOf(Par);
end;

function proc2(var Par: TArray): word;
begin
  Result := SizeOf(Par);
end;

function Proc3(var Par: array[20] of byte): word;
begin
  Result := SizeOf(Par);
end;

function proc4(var Par: string[50]): word;
begin
  Result := SizeOf(Par);
end;

function proc5(var Par: string): word;
begin
  Result := SizeOf(Par);
end;

function proc6(Par: PString): word;
begin
  Result := SizeOf(Par);
end;

function proc7(Par: PArray): word;
begin
```

```
  Result := SizeOf(Par);
end;

procedure Proc8(var Par: array[10] of byte);
begin
  memset(@Par, $ff, SizeOf(Par));
end;

procedure Proc9(var Par: array[30] of byte);
begin
  memset(@Par, $ff, SizeOf(Par));
end;

procedure Proc10(var Par: string[5]);
begin
  Par := Par + 'defghij'; // the tempstring created for the concatenation is only a
string[5] type !!!
  //StrCat(Par, 'defghij');
end;

procedure Proc11(var Par: string);
begin
  Par := Par + 'ghi'; // the tempstring created for the concatenation is only a string[2]
type !!!
end;

procedure Proc14(Var V1: word);
begin
  V1 := $1234;
end;

begin
  // types
  Siz := SizeOf(TArray);   // result = 10  // exact size
  Siz := SizeOf(TString);  // result = 11  // exact size

  // variables
  Siz := SizeOf(Strng);    // result = 5   // exact size
  Siz := SizeOf(Arr);      // result = 25  // exact size

  // function var parameters
  Siz := Proc1(Strng);     // result = 11  <-- size of the formal function parameter
  Siz := proc2(Arr);       // result = 10  <-- size of the formal function parameter
  Siz := Proc3(Arr);       // result = 20  <-- size of the formal function parameter
  Siz := Proc4(Strng);     // result = 51  <-- size of the formal function parameter
  Siz := Proc5(Strng);     // result = 0   <-- unknown size!!!!

  // external items: make sure the compiler can find the "external" unit
  Siz := SizeOf(ExtType);   // result = 100 exact size of external type
  Siz := SizeOf(ExtString); // result = 31  exact size of external variable, no matter
what dummy size defined here :-)
  Siz := SizeOf(ExtArray);  // result = 12  exact size of external variable, no matter
what dummy size defined here :-)
  Siz := SizeOf(Carr);      // result = 7   exact size of external constant, no matter
what dummy size defined here :-)

  // pointers
  Siz := Proc6(@Strng);     // result = 2 <--- size of the pointer itself, not of the
variable it points to
```

```
  Siz := Proc7(@Arr);       // result = 2 <--- size of the pointer itself, not of the
variable it points to

{ Extract from help: If the operand is a parameter declared as array type or function
type, sizeof gives the size of the pointer.
  When applied to records, sizeof gives the total number of bytes, including any padding.
The operator sizeof cannot be applied to a function.
}

  // concatenation tests

  Strng := 'xxxx'; // content to test that it has not changed
  memset(@ExtArray, 0, SizeOf(ExtArray));

  String10 := '';
  Proc10(String10);  // ---> the tempstring created for the concatenation is only a
string[5] type !!!

  String10 := 'abc';
  Proc11(String10);  // ---> the tempstring created for the concatenation is only a
string[2] type !!!

  Strng := 'xxxx'; // content to test that it has not changed
  Siz := 20;       // content to test that it has not changed

  memset(@Arr, 0, SizeOf(Arr)); // Clear Arr
  Proc8(Arr);                   // Fill Arr, <--- Does not crash the program, but only
fills the first 10 bytes of Arr

  memset(@Arr, 0, SizeOf(Arr)); // Clear Arr
  Proc9(Arr);                   // Fill Arr, <--- Does crash the program: will also fill
variables with higher address than Arr also (Siz and Strng)

// Wrong Var parameter test

  Memset(@Arr, 0, SizeOf(Arr));
  Arr[4] := $aa;
  Arr[5] := $bb;
  Proc14(Arr[4]); // The actual parameter is one byte, but both Arr[4] and Arr[5] are
overwritten (Formal parameter is a word).

end.
```

The extra unit in mP:

```
unit ExternalUnit;

Type ExtType = array[100] of byte;

var ExtString: string[30];
    ExtArray: array[12] of byte;

const Carr: array[7] of byte = (1,2,3,4,5,6,7);

implementation

end.
```

7 mB Test/Example Code

The main unit in mB:

```
program SizeOftest

 ' Declarations section

typedef TArray as byte[10]
typedef TString as string[10]
typedef PArray as ^TArray
typedef PString as ^TString

dim
  Siz as word
  Strng as string[4]
  String10 as TString
  ChArr as char[15]
  Arr as byte[25]

  ExtString as string[1] external ' dummy size
  ExtArry as byte[1] external ' dummy size

  I as byte

const
  Carr as byte[1] external

sub function proc1(dim byref Par as Tstring) as word
'begin sub
  Result = SizeOf(Par)
end sub

sub function proc2(dim byref Par as TArray) as word
'begin sub
  Result = SizeOf(Par)
end sub

sub function Proc3(dim byref Par as byte[20]) as word
'begin sub
  Result = SizeOf(Par)
end sub

sub function proc4(dim byref Par as string[50]) as word
'begin sub
  Result = SizeOf(Par)
end sub

sub function proc5(dim byref Par as string) as word
'begin sub
  Result = SizeOf(Par)
end sub

sub function proc6(dim Par as PString) as word
'begin sub
  Result = SizeOf(Par)
end sub
```

```
sub function proc7(dim Par as PArray) as word
'begin sub
  Result = SizeOf(Par)
end sub

sub procedure Proc8(dim byref Par as byte[10])
'begin sub
  memset(@Par, $ff, SizeOf(Par))
end sub

sub procedure Proc9(dim byref Par as byte[30])
'begin sub
  memset(@Par, $ff, SizeOf(Par))
end sub

sub procedure Proc10(dim byref Par as string[5])
'begin sub
  Par = Par + "defghij" ' the tempstring created for the concatenation is only a
string[5] type !!!
  'StrCat(Par, "defghij");
end sub

sub procedure Proc11(dim byref Par as string)
'begin sub
  Par = Par + "ghi" ' the tempstring created for the concatenation is only a string[2]
type !!!
end sub

sub procedure Proc14(dim byref V1 as word)
'begin sub
  V1 = $1234
end sub

main:
  ' types
  Siz = SizeOf(TArray)    ' result = 10  // exact size
  Siz = SizeOf(TString)   ' result = 11  // exact size

  ' variables
  Siz = SizeOf(Strng)     ' result = 5   // exact size
  Siz = SizeOf(Arr)       ' result = 25  // exact size

  ' function var parameters
  Siz = Proc1(Strng)      ' result = 11  <-- size of the formal function parameter
  Siz = proc2(Arr)        ' result = 10  <-- size of the formal function parameter
  Siz = Proc3(Arr)        ' result = 20  <-- size of the formal function parameter
  Siz = Proc4(Strng)      ' result = 51  <-- size of the formal function parameter
  Siz = Proc5(Strng)      ' result = 0   <-- unknown size!!!!

  ' external items: make sure the compiler can find the "external" unit
  Siz = SizeOf(ExtType)    ' result = 100 exact size of external type
  Siz = SizeOf(ExtString) ' result = 31  exact size of external variable, no matter what
dummy size defined here :-)
  Siz = SizeOf(ExtArray)   ' result = 12  exact size of external variable, no matter what
dummy size defined here :-)
  Siz = SizeOf(Carr)       ' result = 7   exact size of external constant, no matter what
dummy size defined here :-)

  ' pointers
  Siz = Proc6(@Strng)      ' result = 2 <--- size of the pointer itself, not of the
variable it points to
  Siz = Proc7(@Arr)        ' result = 2 <--- size of the pointer itself, not of the
variable it points to

' Extract from help: If the operand is a parameter declared as array type or function
type, sizeof gives the size of the pointer.
'  When applied to records, sizeof gives the total number of bytes, including any
padding. The operator sizeof cannot be applied to a function.


  ' concatenation tests

  Strng = "xxxx" ' content to test that it has not changed
  memset(@ExtArray, 0, SizeOf(ExtArray))

  String10 = ""
  Proc10(String10)  ' ---> the tempstring created for the concatenation is only a
string[5] type !!!

  String10 = "abc"
  Proc11(String10)  ' ---> the tempstring created for the concatenation is only a
string[2] type !!!

  Strng = "xxxx" ' content to test that it has not changed
  Siz = 20       ' content to test that it has not changed

  memset(@Arr, 0, SizeOf(Arr)) ' Clear Arr
  Proc8(Arr)                   ' Fill Arr, <--- Does not crash the program, but only
fills the first 10 bytes of Arr

  memset(@Arr, 0, SizeOf(Arr)) ' Clear Arr
  Proc9(Arr)                   ' Fill Arr, <--- Does crash the program: will also fill
variables with higher address than Arr also (Siz and Strng)

' Wrong Var parameter test

  Memset(@Arr, 0, SizeOf(Arr))
  Arr[4] = $aa
  Arr[5] = $bb
  Proc14(Arr[4]) ' The actual parameter is one byte, but both Arr[4] and Arr[5] are
overwritten (Formal parameter is a word).

end.
```

The extra unit in mB:

```
unit ExternalUnit;

Type ExtType = array[100] of byte;

var ExtString: string[30];
    ExtArray: array[12] of byte;

const Carr: array[7] of byte = (1,2,3,4,5,6,7);

implementation
```

```
end.
```

[end of document]