

Re: Fastcode MM B&V 0.41

Source:

<http://coding.derkeiler.com/Archive/Delphi/borland.public.delphi.language.basm/2005-05/msg00823.html>

- *From:* "Dennis" <mariandkc@xxxxxxxxxxxxxxxxxxx>
 - *Date:* Tue, 31 May 2005 18:28:56 +0200
-

Hi

> - HEAP_NO_SERIALIZE doesn't seem to be defined (Delphi 7). Definition:
> const HEAP_NO_SERIALIZE = \$00000001;

I need to add this to my crappy crappy MM ;-)

Regards
Dennis

```
unit DKC_IA32_MM_Unit;
```

```
//Version 0.11 31-5-2005
```

```
interface
```

```
implementation
```

```
uses  
Windows;
```

```
type  
TAllocType = record  
PBlock : Pointer;  
InternalSize : Integer;  
ExternalSize : Integer;  
SmallAlloc : Boolean;  
end;
```

```
TAllocTypeArray = array[0..1000000] of TAllocType;  
PAllocTypeArray = ^TAllocTypeArray;
```

```
var  
Heap: THandle;  
RTLCriticalSection : TRTLCriticalSection;  
LastUsedIndexGlobal : Integer;  
AllocArraySize : integer;  
NoOfLivePointers: integer;
```

```
AllocArray : PAllocTypeArray;
```

```
const
```

```
HEAP_NO_SERIALIZE = $00000001;
ALLOCARRAYINITIALSIZE : Integer = 1000;
SPLITSIZE : Integer = 100*1024*1024;
FLAGS : Cardinal = HEAP_NO_SERIALIZE;
SMALLMOVESIZE = 36;//For JOH Move
SHRINKSIZE : Integer = 200;
GROWSIZE : Integer = 200;
MAXNOOFDEFRAGROUNDS : Integer = 100;
//For upsize
OVERALLOCPERCENTAGESMALLUPSIZESIZE : Double = 2;
OVERALLOCPERCENTAGEBIGUPSIZESIZE : Double = 1.1;
OVERALLOCEXTRA : Integer = 32;
//For downsize
OVERALLOCPERCENTAGESMALLDOWNSIZESIZE : Double = 1/2;
OVERALLOCPERCENTAGEBIGDOWNSIZESIZE : Double = 1/1.1;
```

```
{-----}
{Perform Forward Move of 0..36 Bytes}
{On Entry, ECX = Count, EAX = Source+Count, EDX = Dest+Count. Destroys ECX}
procedure SmallForwardMove_9;
asm
jmp dword ptr [@@FwdJumpTable+ecx*4]
nop {Align Jump Table}
@@FwdJumpTable:
dd @@Done {Removes need to test for zero size move}
dd @@Fwd01, @@Fwd02, @@Fwd03, @@Fwd04, @@Fwd05, @@Fwd06, @@Fwd07,
@@Fwd08
dd @@Fwd09, @@Fwd10, @@Fwd11, @@Fwd12, @@Fwd13, @@Fwd14, @@Fwd15,
@@Fwd16
dd @@Fwd17, @@Fwd18, @@Fwd19, @@Fwd20, @@Fwd21, @@Fwd22, @@Fwd23,
@@Fwd24
dd @@Fwd25, @@Fwd26, @@Fwd27, @@Fwd28, @@Fwd29, @@Fwd30, @@Fwd31,
@@Fwd32
dd @@Fwd33, @@Fwd34, @@Fwd35, @@Fwd36
@@Fwd36:
mov ecx, [eax-36]
mov [edx-36], ecx
@@Fwd32:
mov ecx, [eax-32]
mov [edx-32], ecx
@@Fwd28:
mov ecx, [eax-28]
mov [edx-28], ecx
@@Fwd24:
mov ecx, [eax-24]
mov [edx-24], ecx
@@Fwd20:
mov ecx, [eax-20]
```

```
mov [edx-20], ecx
@@Fwd16:
mov ecx, [eax-16]
mov [edx-16], ecx
@@Fwd12:
mov ecx, [eax-12]
mov [edx-12], ecx
@@Fwd08:
mov ecx, [eax-8]
mov [edx-8], ecx
@@Fwd04:
mov ecx, [eax-4]
mov [edx-4], ecx
ret
nop
@@Fwd35:
mov ecx, [eax-35]
mov [edx-35], ecx
@@Fwd31:
mov ecx, [eax-31]
mov [edx-31], ecx
@@Fwd27:
mov ecx, [eax-27]
mov [edx-27], ecx
@@Fwd23:
mov ecx, [eax-23]
mov [edx-23], ecx
@@Fwd19:
mov ecx, [eax-19]
mov [edx-19], ecx
@@Fwd15:
mov ecx, [eax-15]
mov [edx-15], ecx
@@Fwd11:
mov ecx, [eax-11]
mov [edx-11], ecx
@@Fwd07:
mov ecx, [eax-7]
mov [edx-7], ecx
mov ecx, [eax-4]
mov [edx-4], ecx
ret
nop
@@Fwd03:
movzx ecx, word ptr [eax-3]
mov [edx-3], cx
movzx ecx, byte ptr [eax-1]
mov [edx-1], cl
ret
@@Fwd34:
mov ecx, [eax-34]
```

```
mov [edx-34], ecx
@@Fwd30:
mov ecx, [eax-30]
mov [edx-30], ecx
@@Fwd26:
mov ecx, [eax-26]
mov [edx-26], ecx
@@Fwd22:
mov ecx, [eax-22]
mov [edx-22], ecx
@@Fwd18:
mov ecx, [eax-18]
mov [edx-18], ecx
@@Fwd14:
mov ecx, [eax-14]
mov [edx-14], ecx
@@Fwd10:
mov ecx, [eax-10]
mov [edx-10], ecx
@@Fwd06:
mov ecx, [eax-6]
mov [edx-6], ecx
@@Fwd02:
movzx ecx, word ptr [eax-2]
mov [edx-2], cx
ret
nop
nop
nop
@@Fwd33:
mov ecx, [eax-33]
mov [edx-33], ecx
@@Fwd29:
mov ecx, [eax-29]
mov [edx-29], ecx
@@Fwd25:
mov ecx, [eax-25]
mov [edx-25], ecx
@@Fwd21:
mov ecx, [eax-21]
mov [edx-21], ecx
@@Fwd17:
mov ecx, [eax-17]
mov [edx-17], ecx
@@Fwd13:
mov ecx, [eax-13]
mov [edx-13], ecx
@@Fwd09:
mov ecx, [eax-9]
mov [edx-9], ecx
@@Fwd05:
```

```

mov ecx, [eax-5]
mov [edx-5], ecx
@@Fwd01:
movzx ecx, byte ptr [eax-1]
mov [edx-1], cl
ret
@@Done:
end; {SmallForwardMove}

```

```

{-----}
{Perform Backward Move of 0..36 Bytes}
{On Entry, ECX = Count, EAX = Source, EDX = Dest. Destroys ECX}
procedure SmallBackwardMove_9;
asm
jmp dword ptr [@@BwdJumpTable+ecx*4]
nop {Align Jump Table}
@@BwdJumpTable:
dd @@Done {Removes need to test for zero size move}
dd @@Bwd01, @@Bwd02, @@Bwd03, @@Bwd04, @@Bwd05, @@Bwd06, @@Bwd07,
@@Bwd08
dd @@Bwd09, @@Bwd10, @@Bwd11, @@Bwd12, @@Bwd13, @@Bwd14, @@Bwd15,
@@Bwd16
dd @@Bwd17, @@Bwd18, @@Bwd19, @@Bwd20, @@Bwd21, @@Bwd22, @@Bwd23,
@@Bwd24
dd @@Bwd25, @@Bwd26, @@Bwd27, @@Bwd28, @@Bwd29, @@Bwd30, @@Bwd31,
@@Bwd32
dd @@Bwd33, @@Bwd34, @@Bwd35, @@Bwd36
@@Bwd36:
mov ecx, [eax+32]
mov [edx+32], ecx
@@Bwd32:
mov ecx, [eax+28]
mov [edx+28], ecx
@@Bwd28:
mov ecx, [eax+24]
mov [edx+24], ecx
@@Bwd24:
mov ecx, [eax+20]
mov [edx+20], ecx
@@Bwd20:
mov ecx, [eax+16]
mov [edx+16], ecx
@@Bwd16:
mov ecx, [eax+12]
mov [edx+12], ecx
@@Bwd12:
mov ecx, [eax+8]
mov [edx+8], ecx
@@Bwd08:
mov ecx, [eax+4]
mov [edx+4], ecx

```

```
@@Bwd04:
mov ecx, [eax]
mov [edx], ecx
ret
nop
nop
nop
@@Bwd35:
mov ecx, [eax+31]
mov [edx+31], ecx
@@Bwd31:
mov ecx, [eax+27]
mov [edx+27], ecx
@@Bwd27:
mov ecx, [eax+23]
mov [edx+23], ecx
@@Bwd23:
mov ecx, [eax+19]
mov [edx+19], ecx
@@Bwd19:
mov ecx, [eax+15]
mov [edx+15], ecx
@@Bwd15:
mov ecx, [eax+11]
mov [edx+11], ecx
@@Bwd11:
mov ecx, [eax+7]
mov [edx+7], ecx
@@Bwd07:
mov ecx, [eax+3]
mov [edx+3], ecx
mov ecx, [eax]
mov [edx], ecx
ret
nop
nop
nop
@@Bwd03:
movzx ecx, word ptr [eax+1]
mov [edx+1], cx
movzx ecx, byte ptr [eax]
mov [edx], cl
ret
nop
nop
@@Bwd34:
mov ecx, [eax+30]
mov [edx+30], ecx
@@Bwd30:
mov ecx, [eax+26]
mov [edx+26], ecx
```

```
@@Bwd26:
mov ecx, [eax+22]
mov [edx+22], ecx
@@Bwd22:
mov ecx, [eax+18]
mov [edx+18], ecx
@@Bwd18:
mov ecx, [eax+14]
mov [edx+14], ecx
@@Bwd14:
mov ecx, [eax+10]
mov [edx+10], ecx
@@Bwd10:
mov ecx, [eax+6]
mov [edx+6], ecx
@@Bwd06:
mov ecx, [eax+2]
mov [edx+2], ecx
@@Bwd02:
movzx ecx, word ptr [eax]
mov [edx], cx
ret
nop
@@Bwd33:
mov ecx, [eax+29]
mov [edx+29], ecx
@@Bwd29:
mov ecx, [eax+25]
mov [edx+25], ecx
@@Bwd25:
mov ecx, [eax+21]
mov [edx+21], ecx
@@Bwd21:
mov ecx, [eax+17]
mov [edx+17], ecx
@@Bwd17:
mov ecx, [eax+13]
mov [edx+13], ecx
@@Bwd13:
mov ecx, [eax+9]
mov [edx+9], ecx
@@Bwd09:
mov ecx, [eax+5]
mov [edx+5], ecx
@@Bwd05:
mov ecx, [eax+1]
mov [edx+1], ecx
@@Bwd01:
movzx ecx, byte ptr[eax]
mov [edx], cl
ret
```

```

nop
nop
@@Done:
end; {SmallBackwardMove}

```

```

{-----}
{Move ECX Bytes from EAX to EDX, where EAX > EDX and ECX > 36
(SMALLMOVESIZE)}
procedure Forwards_IA32_9;
asm
push edx
fild qword ptr [eax]
lea eax, [eax+ecx-8]
lea ecx, [ecx+edx-8]
fild qword ptr [eax]
push ecx
neg ecx
and edx, -8
lea ecx, [ecx+edx+8]
pop edx
@FwdLoop:
fild qword ptr [eax+ecx]
fistp qword ptr [edx+ecx]
add ecx, 8
jl @FwdLoop
fistp qword ptr [edx]
pop edx
fistp qword ptr [edx]
end; {Forwards_IA32}

```

```

{-----}
{Move ECX Bytes from EAX to EDX, where EAX < EDX and ECX > 36
(SMALLMOVESIZE)}
procedure Backwards_IA32_9;
asm
sub ecx, 8
push ecx
fild qword ptr [eax+ecx] {Last 8}
fild qword ptr [eax] {First 8}
add ecx, edx
and ecx, -8
sub ecx, edx
@BwdLoop:
fild qword ptr [eax+ecx]
fistp qword ptr [edx+ecx]
sub ecx, 8
jg @BwdLoop
pop ecx
fistp qword ptr [edx] {First 8}
fistp qword ptr [edx+ecx] {Last 8}
end; {Backwards_IA32}

```

```

{-----}
{Move using IA32 Instruction Set Only}
procedure MoveJOH_IA32_9(const Source; var Dest; Count : Integer);
asm
  cmp ecx, SMALLMOVESIZE
  ja @Large {Count > SMALLMOVESIZE or Count < 0}
  cmp eax, edx
  jbe @SmallCheck
  add eax, ecx
  add edx, ecx
  jmp SmallForwardMove_9
  @SmallCheck:
  jne SmallBackwardMove_9
  ret {For Compatibility with Delphi's move for Source = Dest}
  @Large:
  jng @Done {For Compatibility with Delphi's move for Count < 0}
  cmp eax, edx
  ja Forwards_IA32_9
  je @Done {For Compatibility with Delphi's move for Source = Dest}
  sub edx, ecx
  cmp eax, edx
  lea edx, [edx+ecx]
  jna Forwards_IA32_9
  jmp Backwards_IA32_9 {Source/Dest Overlap}
  @Done:
end; {MoveJOH_IA32}

procedure InitializeAllocArray(StartIndex, StopIndex : Integer);
var
  Index : Integer;

begin
  { if (StartIndex < 0) or (StartIndex > AllocArraySize-1) then
  RunError(203);
  if (StopIndex < 0) or (StopIndex > AllocArraySize-1) then
  RunError(203);}
  for Index := StartIndex to StopIndex do
  begin
    AllocArray[Index].PBlock := nil;
    //AllocArray[Index].Active := False;
    AllocArray[Index].InternalSize := 0;
    AllocArray[Index].ExternalSize := 0;
    AllocArray[Index].SmallAlloc := True
  end;
end;

function GrowAllocArray(var AllocArray : PAllocTypeArray; OldSize : Integer)
: Integer;
var
  NewSize : Integer;

```

```

begin
NewSize := OldSize + GROWSIZE;
AllocArray := HeapRealloc(Heap, FLAGS, AllocArray , NewSize *
SizeOf(TAllocType));
if AllocArray = nil then
begin
//Try with a smaller GROWSIZE
NewSize := OldSize + 1;
AllocArray := HeapRealloc(Heap, FLAGS, AllocArray , NewSize *
SizeOf(TAllocType));
if AllocArray <> nil then
begin
InitializeAllocArray(OldSize, NewSize-1);
Result := NewSize;
end
else
begin
RunError(203);
Result := 0;//For compiler
end;
end
else
begin
InitializeAllocArray(OldSize, NewSize-1);
Result := NewSize;
end;
end;

function ShrinkAllocArray(var AllocArray : PAllocTypeArray; OldSize :
Integer) : Integer;
var
HighIndex, NewSize, Index, LowIndex, NoOfDefragRounds : Integer;

begin
NoOfDefragRounds := 0;
HighIndex := AllocArraySize;
LowIndex := 0;
repeat
//Find highest used index
repeat
Dec(HighIndex);
until(AllocArray[HighIndex].PBlock <> nil);
//Find lowest unused index
repeat
Inc(LowIndex);
until(AllocArray[LowIndex].PBlock = nil);
if LowIndex < HighIndex then
begin
//Copy HighIndex to LowIndex
AllocArray[LowIndex].PBlock := AllocArray[HighIndex].PBlock;

```

```

AllocArray[LowIndex].InternalSize := AllocArray[HighIndex].InternalSize;
AllocArray[LowIndex].ExternalSize := AllocArray[HighIndex].ExternalSize;
AllocArray[LowIndex].SmallAlloc := AllocArray[HighIndex].SmallAlloc;
//Clear HighIndex
AllocArray[HighIndex].PBlock := nil;
AllocArray[HighIndex].InternalSize := 0;
AllocArray[HighIndex].ExternalSize := 0;
AllocArray[HighIndex].SmallAlloc := True
end
else
//No more to defrag
Break;
Inc(NoOfDefragRounds)
until(NoOfDefragRounds >= MAXNOOFDEFRAGROUNDS);
//Find highest used index
Index := HighIndex;
repeat
Dec(Index);
until(AllocArray[Index].PBlock <> nil);
NewSize := OldSize - SHRINKSIZE;
//Do not shrink below any used entries
if NewSize < Index+1 then
NewSize := Index+1;
if NewSize < OldSize then
begin
AllocArray := HeapRealloc(Heap, FLAGS, AllocArray , NewSize *
SizeOf(TAllocType));
LastUsedIndexGlobal := NewSize-1;
end;
if AllocArray = nil then
RunError(203);
Result := NewSize;
end;

function GetSize(Index : Integer) : Integer; overload;
begin
{if (Index < 0) or (Index > AllocArraySize-1) then
begin
Result := 0; //For compiler
RunError(203);
Exit;
end;}
Result := AllocArray[Index].InternalSize;
end;

function GetExternalSize(Index : Integer) : Integer;
begin
{if (Index < 0) or (Index > AllocArraySize-1) then
begin
Result := 0; //For compiler
RunError(203);

```

```

Exit;
end;}
Result := AllocArray[Index].ExternalSize;
end;

procedure SetSize(Index, NewSize, ExternalSize : Integer);
begin
{ if (Index < 0) or (Index > AllocArraySize-1) then
begin
RunError(203);
Exit;
end;}
AllocArray[Index].InternalSize := NewSize;
AllocArray[Index].ExternalSize := ExternalSize;
end;

function GetSmallAlloc(Index : Integer) : Boolean; overload;
begin
{ if (Index < 0) or (Index > AllocArraySize-1) then
begin
Result := True; //For compiler
RunError(203);
Exit;
end;}
Result := AllocArray[Index].SmallAlloc;
end;

procedure AddToAllocTypeArray(P : Pointer; SmallAlloc : Boolean; Size,
ExternalSize : Integer);
var
Index1, Index2 : Integer;

begin
Inc(NoOfLivePointers);
if NoOfLivePointers > AllocArraySize then
AllocArraySize := GrowAllocArray(AllocArray, AllocArraySize);
Index1 := LastUsedIndexGlobal;
Index2 := LastUsedIndexGlobal+1;
if Index1 < 0 then
Index1 := 0;
if Index2 > AllocArraySize-1 then
Index2 := AllocArraySize-1;
repeat
//if (AllocArray[Index1].Active = False) then
if (AllocArray[Index1].PBlock = nil) then
begin
LastUsedIndexGlobal := Index1;
AllocArray[Index1].PBlock := P;
AllocArray[Index1].InternalSize := Size;
AllocArray[Index1].ExternalSize := ExternalSize;
AllocArray[Index1].SmallAlloc := SmallAlloc;

```

```

Exit;
end;
if (AllocArray[Index2].PBlock = nil) then
begin
LastUsedIndexGlobal := Index2;
AllocArray[Index2].PBlock := P;
AllocArray[Index2].InternalSize := Size;
AllocArray[Index2].ExternalSize := ExternalSize;
AllocArray[Index2].SmallAlloc := SmallAlloc;
Exit;
end;
Dec(Index1);
Inc(Index2);
if (Index1 < 0) and (Index2 > AllocArraySize-1) then
begin
//Did not find space for pointer
RunError(203);
Exit;
end;
if Index1 < 0 then
Index1 := 0;
if Index2 > AllocArraySize-1 then
Index2 := AllocArraySize-1;
until(False);
end;

procedure RemoveFromAllocTypeArray(Index : Integer); overload;
begin
{ if (Index < 0) or (Index > AllocArraySize-1) then
begin
RunError(203);
end;}
AllocArray[Index].PBlock := nil;
AllocArray[Index].InternalSize := 0;
AllocArray[Index].ExternalSize := 0;
AllocArray[Index].SmallAlloc := True;
Dec(NoOfLivePointers);
if NoOfLivePointers < AllocArraySize-SHRINKSIZE then
AllocArraySize := ShrinkAllocArray(AllocArray, AllocArraySize);
end;

function GetIndex(P : Pointer) : Integer;
var
Index1, Index2 : Integer;

begin
if LastUsedIndexGlobal > AllocArraySize-2 then
LastUsedIndexGlobal := AllocArraySize-2;
Index1 := LastUsedIndexGlobal;
Index2 := LastUsedIndexGlobal+1;
{ if Index1 < 0 then

```

```

Index1 := 0;
if Index2 > AllocArraySize-1 then
Index2 := AllocArraySize-1;}
repeat
if (AllocArray[Index1].PBlock = P) then
begin
LastUsedIndexGlobal := Index1;
Result := Index1;
Exit;
end;
if (AllocArray[Index2].PBlock = P) then
begin
LastUsedIndexGlobal := Index2;
Result := Index2;
Exit;
end;
Dec(Index1);
Inc(Index2);
if (Index1 < 0) and (Index2 > AllocArraySize-1) then
begin
//Did not find pointer
RunError(203);
Result := -1;
Exit;
end;
if Index1 < 0 then
Index1 := 0;
if Index2 > AllocArraySize-1 then
Index2 := AllocArraySize-1;
until(False);
end;

function DKCGetMem(Size: Integer): Pointer;
var
OverSize : Integer;

begin
if IsMultiThread then
EnterCriticalSection(RTLCriticalSection);
if Size < SPLITSIZE then
begin
OverSize := Size + OVERALLOCEXTRA;
Result := HeapAlloc(Heap, FLAGS, OverSize);
AddToAllocTypeArray(Result, True, OverSize, Size);
end
else
begin
Result := VirtualAlloc(nil, Size, MEM_COMMIT+MEM_TOP_DOWN,
PAGE_READWRITE);
AddToAllocTypeArray(Result, False, Size, Size);
end;
end;

```

```
if IsMultiThread then
LeaveCriticalSection(RTLCriticalSection);
end;

function DKCFreeMem(Ptr: Pointer): Integer;
var
Res : Boolean;
Index : Integer;

begin
if IsMultiThread then
EnterCriticalSection(RTLCriticalSection);
Index := GetIndex(Ptr);
if GetSmallAlloc(Index) then
begin
if HeapFree(Heap, FLAGS, Ptr) then
begin
RemoveFromAllocTypeArray(Index);
Result := 0;
if HeapCompact(Heap, FLAGS) = 0 then
RunError(203);
end
else
Result := 1;
end
else
begin
Res := VirtualFree(Ptr, 0, MEM_RELEASE);
if Res then
begin
RemoveFromAllocTypeArray(Index);
Result := 0;
end
else
Result := 1;
end;
if IsMultiThread then
LeaveCriticalSection(RTLCriticalSection);
end;

function DKCReallocMem(Ptr: Pointer; Size: Integer): Pointer;
var
OldIndex, OldSize, NewSize, OldExternalSize, NewOverSize : Integer;
NewPtr, OldPtr : Pointer;

begin
if IsMultiThread then
EnterCriticalSection(RTLCriticalSection);
NewSize := Size;
OldPtr := Ptr;
OldIndex := GetIndex(Ptr);
```

```

OldSize := GetSize(OldIndex);
if GetSmallAlloc(OldIndex) then
begin
if NewSize < SPLITSIZE then
begin
//Realloc small as small
if (NewSize > OldSize) then //Upsize
begin
//Alloc more than requested
NewOverSize := Round(NewSize * OVERALLOCPERCENTAGESMALLUPSIZ) +
OVERALLOCEXTRA;
Result := HeapRealloc(Heap, FLAGS, OldPtr, NewOverSize);
end
else if (NewSize < Round(OldSize * OVERALLOCPERCENTAGESMALLDOWNSIZ) -
OVERALLOCEXTRA) then //Downsize
begin
//Allocate requested size
NewOverSize := NewSize;
Result := HeapRealloc(Heap, FLAGS, OldPtr, NewOverSize);
end
else
begin
//OverSize did not change because no realloc took place
NewOverSize := OldSize;
Result := OldPtr;
end;
if Result = OldPtr then
begin
SetSize(OldIndex, NewOverSize, NewSize);
end
else
begin
RemoveFromAllocTypeArray(OldIndex);
AddToAllocTypeArray(Result, True, NewOverSize, NewSize);
end;
end
else
begin
//Realloc small as big
//Get new block
Result := VirtualAlloc(nil, NewSize, MEM_COMMIT+MEM_TOP_DOWN,
PAGE_READWRITE);
NewPtr := Result;
if Result <> nil then
begin
AddToAllocTypeArray(NewPtr, False, NewSize, NewSize);
OldExternalSize := GetExternalSize(OldIndex);
MoveJOH_IA32_9(OldPtr^, NewPtr^, OldExternalSize);
if not HeapFree(Heap, FLAGS, OldPtr) then
RunError(203)
else

```



```

else
begin
//Realloc failed. Try get a new block
Result := VirtualAlloc(nil, NewOverSize, MEM_COMMIT+MEM_TOP_DOWN,
PAGE_READWRITE);
NewPtr := Result;
if Result <> nil then
begin
AddToAllocTypeArray(NewPtr, False, NewOverSize, NewSize);
OldExternalSize := GetExternalSize(OldIndex);
MoveJOH_IA32_9(OldPtr^, NewPtr^, OldExternalSize);
if not VirtualFree(OldPtr, 0, MEM_RELEASE) then
RunError(203)
else
RemoveFromAllocTypeArray(OldIndex);
end
else
RunError(203);
end;
end
else
begin
//Realloc big as small
Result := HeapAlloc(Heap, FLAGS, NewSize);
NewPtr := Result;
AddToAllocTypeArray(NewPtr, True, NewSize, NewSize);
Move(OldPtr^, NewPtr^, NewSize);
if not VirtualFree(OldPtr, 0, MEM_RELEASE) then
RunError(203)
else
RemoveFromAllocTypeArray(OldIndex);
end;
end;
if IsMultiThread then
LeaveCriticalSection(RTLCriticalSection);
end;

procedure InitMemoryManager;
resourcestring
sError = 'DKC_IA32_MM_Unit must be first unit used by the project';
var
MemMgr: TMemoryManager;

begin
NoOfLivePointers := 0;
Heap := HeapCreate(FLAGS, 0, 0);
if Heap = 0 then
RunError(203); // out of memory
Assert(AllocMemCount = 0, sError);
MemMgr.GetMem := DKCGetMem;
MemMgr.FreeMem := DKCFreeMem;

```

```
MemMgr.ReallocMem := DKCReallocMem;  
SetMemoryManager(MemMgr);  
InitializeCriticalSection(RTLCriticalSection);  
EnterCriticalSection(RTLCriticalSection);  
AllocArraySize := GROWSIZE;  
AllocArray := HeapAlloc(Heap, FLAGS, AllocArraySize * SizeOf(TAllocType));  
InitializeAllocArray(0, AllocArraySize-1);  
LeaveCriticalSection(RTLCriticalSection);  
end;
```

initialization

```
InitMemoryManager;
```

finalization

```
if Heap <> 0 then  
HeapDestroy(Heap);  
DeleteCriticalSection(RTLCriticalSection);
```

end.

.

• **References:**

- ◆ **Fastcode MM B&V 0.41**
 ◇ From: Dennis
- ◆ **Re: Fastcode MM B&V 0.41**
 ◇ From: Avatar Zondertau

- Prev by Date: **Re: Fastcode MM B&V 0.41**
- Next by Date: **Re: Fastcode UpperCase B&V 2.8**
- Previous by thread: **Re: Fastcode MM B&V 0.41 (extended)**
- Next by thread: **Re: Fastcode MM B&V 0.41**
- Index(es):
 - ◆ **Date**
 - ◆ **Thread**