

BRNO UNIVERSITY OF TECHNOLOGY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF CONTROL AND INSTRUMENTATION

Kolejní 4, 616 00 Brno

tel.: +420 5 4114 1113 fax: +420 5 4114 1123

E-mail: kucera@feec.vutbr.cz, <http://taceo.eu>



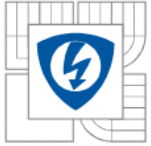
Operační Systémy Reálného Času

VIII.

Pavel Kučera

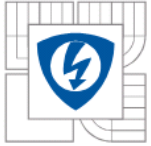
CAK, E112

kucera@feec.vutbr.cz

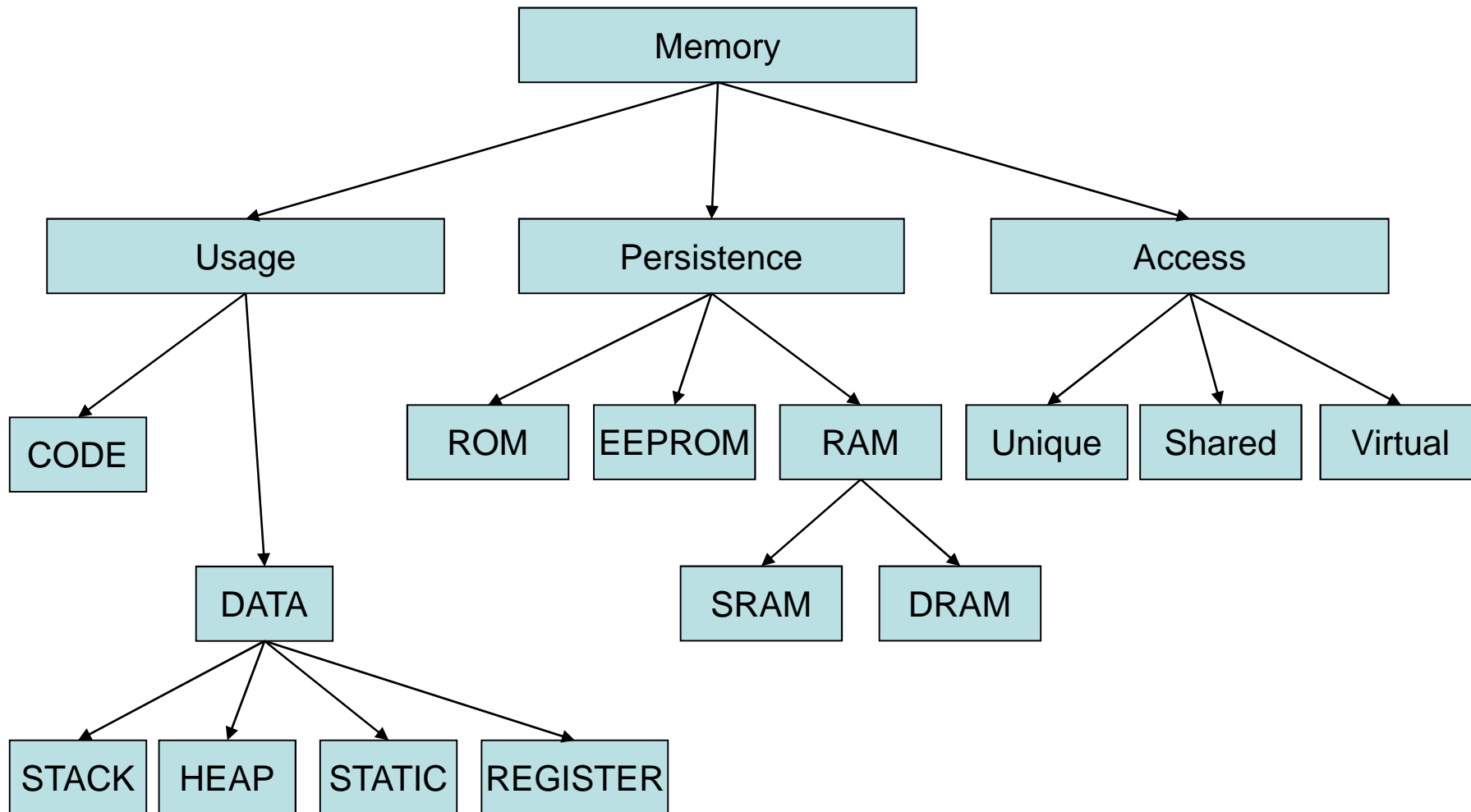


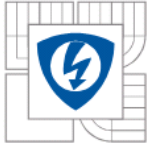
Obsah

1. Správa paměti v RTOS
2. DMA
3. Dynamická alokace



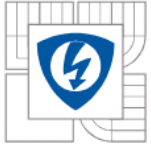
Správa paměti v RTOS





Správa paměti v RTOS

- Stack** real-time odezva, nutno dát pozor na přetečení (LIFO struktura), **znalost překladače, reentrantnost funkcí**
- Static** není nikdy dostatek, zpomalení způsobeno segmentací a stránkováním, lze dosáhnout real-time odezvu. Problémy v RTOS vznikají při použití mechanismu virtualizace paměti.
- Heap** dynamická paměť, kterou proces vyžaduje za běhu. Častá alokace a dealokace způsobují fragmentaci a tím zásadní problém s real-time odezvou.



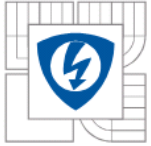
Intel x86 – memory page

page

(stránka) je blok fyzické paměti o určité délce (4 MB), který je využit jako interface mezi fyzickou pamětí a externím úložištěm – nejčastěji HDD.

page fault

je přerušení (vyjímka) generovaná v MMU (Memory Management Unit) procesoru, když proces přistupuje do adresního prostoru, který není nahrán ve fyzické paměti.

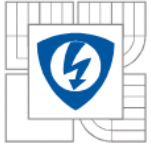


Zabránění page fault

V RTSS je každá paměť a proces implicitně chráněn proti mechanismu stránkování.

Ve Win32 je v případě nutnosti možné zabránit stránkování jednou z následujících funkcí:

RtLockProcess	uzamkne všechny oblasti ve virtuálním adresním prostoru procesu ve fyzické paměti.
RtUnlockProcess	odemkne všechny uzamčené oblasti paměti procesu a umožní tak jejich stránkování.
RtCommitLockStack	vymezí pro zásobník oblast paměti ve fyzické paměti a zabrání jejímu stránkování.
RtCommitLockProcess	Heap vymezí pro haldu procesu oblast paměti ve fyzické paměti a zabrání jejímu stránkování.



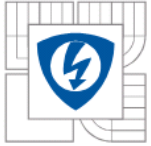
Alokace chráněné paměti v RTX

RtAllocateLockedMemory

Alokuje paměť v nestránkované oblasti fyzické paměti (max. 256 MB). Tato paměť je alokována během startu OS a je hodně fragmentována. Aby byl process v RTX při alokaci této paměti úspěšný, měl by se spustit při boot sekvenci. Funkce vrátí pointer do virtuálního adresního prostoru procesu.

RtFreeLockedMemory

Uvolní před paměť alokovanou v nestránkované oblasti OS.



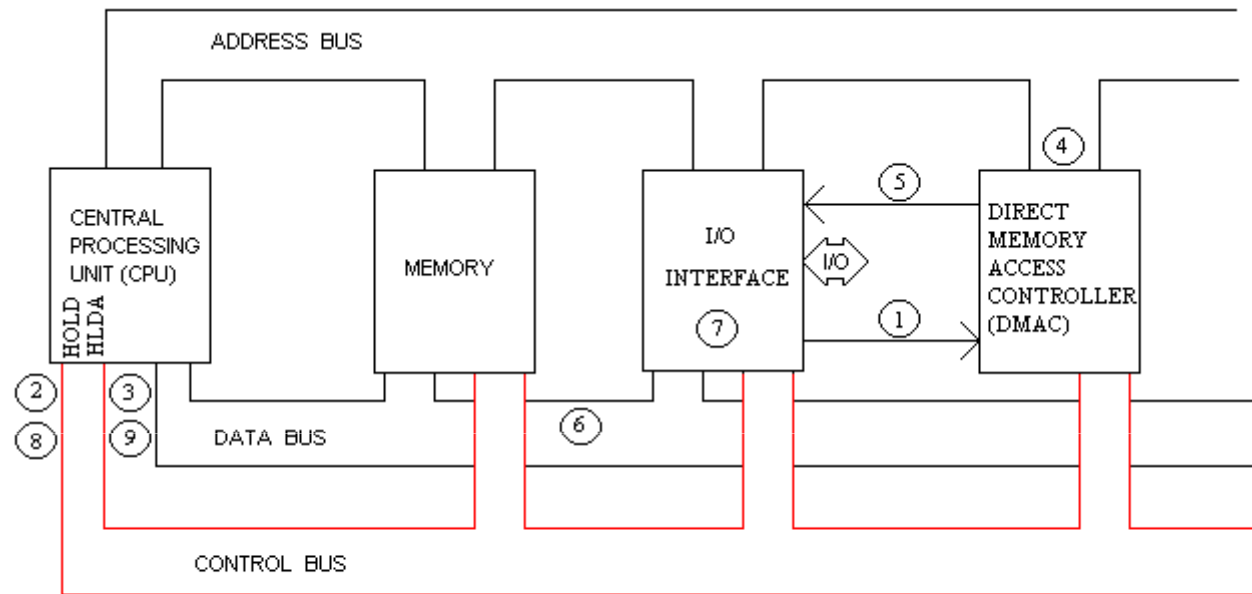
Alokace chráněné paměti v RTX

```
// virtual memory address returned by RtAllocateLockedMemory
static PVOID vAddress;

void main(void) {
    ULONG size=4096; // bytes to allocate
    // Allocate memory
    vAddress = RtAllocateLockedMemory(size);
    if (!vAddress) {
        printf("\nFailure on RtAllocateLockedMemory\t");
        printf("Error=%d\n", GetLastError());
        return; // if this fails - no use to continue
    }
    else {
        printf("\nSuccess on RtAllocateLockedMemory\n");
        printf("Virtual memory address = 0x%08X\n", vAddress);
        // Write to the memory
        *vAddress= 0;
    }
    // Free memory
    if (!RtFreeLockedMemory(vAddress)) {
        printf("\nFailure on RtFreeLockedMemory(0x%08X).\t", vAddress);
        printf("Error Code = %d\n", GetLastError());
    }
    else {
        printf("Success on RtFreeLockedMemory(0x%X).\n", vAddress);
    }
    ExitProcess(0);
}
```



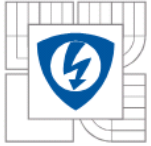
DMA – přímý přístup do paměti



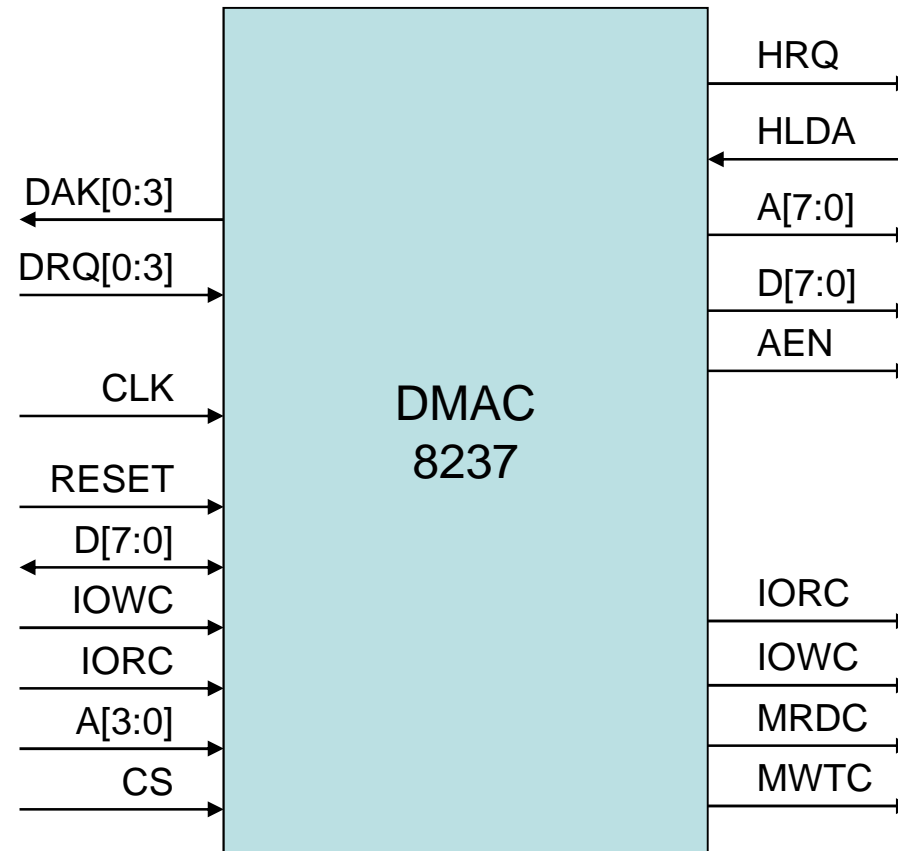
DATA TRANSFER WITH A DMA CONTROLLER

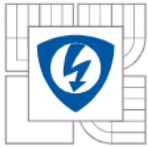
source: www.compeng.dit.ie

1. I/O zařízení žádá o DMA přenos dat.
2. DMA žádá signálem HOLD = 1 o přidělení sběrnice.
3. CPU potvrdí DMA řadiči přidělení sběrnice signálem HLDA = 1.
4. DMA řadič adresuje sběrnici.
5. DMA potvrdí I/O zařízení přenos dat.
6. Data jsou I/O zařízením vložena na datovou sběrnici.
7. Data jsou I/O zařízením potvrzena. Přesun dat byl ukončen.
8. DMA řadič vrátí řízení sběrnice, HOLD = 0.
9. CPU potvrdí převzetí sběrnice, HLDA = 0.
10. Celý cyklus 1 - 9 se opakuje tak dlouho, dokud není přesunut celý blok.



DMA – přímý přístup do paměti





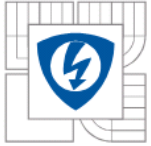
DMA – přímý přístup do paměti

Example of transfer mode

Transfer Mode	Timing Chart	Operation
Single		When DMARQ is received, DMA transfer is executed once, and the bus mastership is released to the CPU. If DMARQ is input again, DMA transfer is executed again. This operation is repeated as long as DMARQ is input and until TC/END is reached.
Single step		When DMARQ is received, DMA transfer and the bus cycle of the CPU are alternately repeated until TC/END is reached.
Demand		This mode seems similar to the single-transfer mode. However, only DMA transfer is executed and the bus mastership is not released to the CPU as long as DMARQ is input. When DMARQ is no longer input, the bus mastership is released to the CPU.
Block (burst)		This mode seems similar to the single-step transfer mode. When DMARQ is input, however, only DMA transfer is executed and the bus mastership is not released to the CPU, until TC/END is reached.

DMARQ : DMA request(peripheral → DMAC)
HLDRQ : Hold request (DMAC → CPU)
HLDAAK : Hold acknowledge (CPU → DMAC)
DMAAK : DMA acknowledge (DMAC → peripheral)

source: http://www.necel.com/en/faq/f_tech.html



DMA v PC

MRDC - *Memory Read Command*

DMAC indikuje paměti, že má poskytnout data.

MWTC - *Memory Write Command*

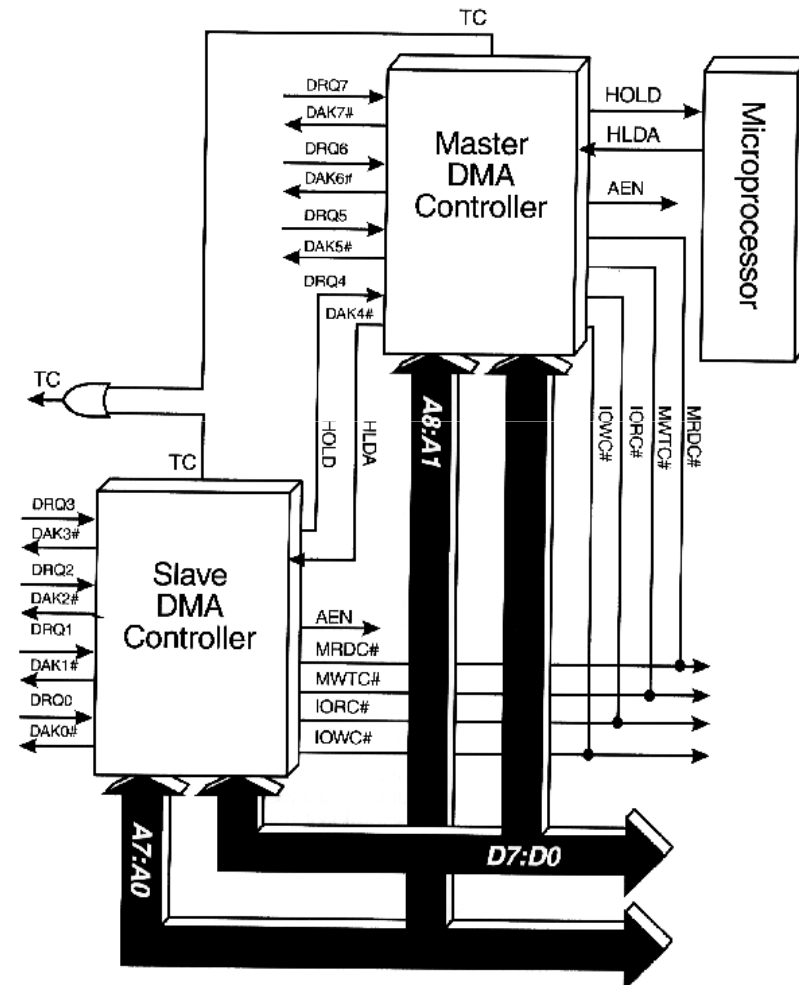
DMAC indikuje paměti, že má zapsat data.

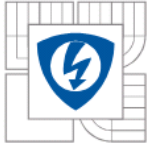
IORC - *IO Read Command*

DMAC indikuje IO zařízení, že má poskytnout data.

IOWC - *IO Write Command*

DMAC indikuje IO zařízení, že má zapsat data.

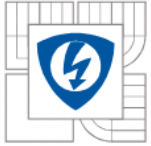




DMA v PC

Standardní přiřazení DMA kanálů v PC

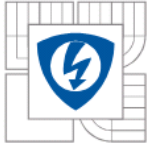
0	16-bit	8-bit	none
1	8/16-bit	8-bit	Some sound cards
2	8/16-bit	8-bit	Floppy disk controller
3	8/16-bit	8-bit	LPT1: in ECP mode
4	none	16-bit	DMA controller cascade
5	16-bit	16-bit	Some sound cards; ISA SCSI host adapter
6	16-bit	16-bit	ISA SCSI host adapter
7	16-bit	16-bit	Some sound cards; ISA SCSI host adapter



DMA v PC

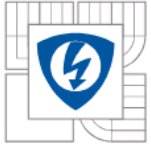
DMA Controller	Slave				Master			
DRQ#	0	1	2	3	4	5	6	7
Memory Address Register I/O Address	00h	02h	04h	06h	C0h	C4h	C8h	CCh
Page Register	87h	83h	81h	82h	(none)	8Bh	89h	8Ah
Count Register I/O Address	01h	03h	05h	07h	(none)	C6h	CAh	CEh

DMA Controller	Slave	Master
Control Register I/O Address	08h	D0h
Mode Register I/O Address	0Bh	D6h
Mask Register I/O Address	0Ah	D4h
Clear Byte F/F I/O Address	0Ch	D8h



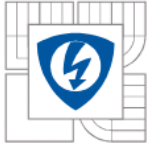
DMA – přímý přístup do paměti

<http://www.cast-inc.com/cores/c8237/index.shtml>



Alokace spojitého bloku v RTX

RtAllocateContiguousMemory	alokuje spojitou oblast ve fyzické paměti a namapuje ji ve virtuálním adresovacím prostoru procesu.
RtFreeContiguousMemory	uvolní spojitou oblast ve fyzické paměti.
RtGetPhysicalAddress	vrátí adresu ve fyzické paměti kam je alokovan spojitý blok.
RtMapMemory	namapuje oblast ve fyzické paměti do virtuálního adresního prostoru procesu v RTX
RtUnmapMemory	odmapuje oblast ve fyzické paměti do virtuálního adresního prostoru procesu v RTX



RtMapMemory

RtMapMemory namapuje oblast ve fyzické paměti do virtuálního adresního prostoru procesu v RTX

Prototype

PVOID

RtMapMemory(

LARGE_INTEGER *physAddr*,

ULONG *Length*,

BOOLEAN *CacheEnable*

);

Parameters

physAddr

LARGE_INTEGER určuje bázovou adresu fyzické paměti určené k namapování do virtuálního adresního prostoru procesu.

Length

32-bit slovo obsahující délku bloku paměti.

CacheEnable

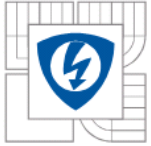
určuje, zda-li Windows použijí k přístupu k tomuto bloku paměti cache paměť.



RTOS - IX



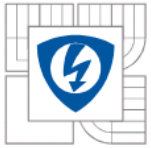
```
static PVOID vAddress;           // virtual memory address returned by RtAllocateContiguousMemory
static LARGE_INTEGER pAddress;   // physical memory address returned by
void main(void) {
LARGE_INTEGER maxPhyAddr; //highest physical memory address
ULONG size=0; //bytes to allocate
maxPhyAddr.QuadPart = 0xFFFFFFFF;
size= 0x1000; // Size in KB
vAddress = RtAllocateContiguousMemory( size, maxPhyAddr);           // Allocate memory
if (!vAddress) {
    printf("\nFailure on RtAllocateContiguousMemory\t");
    printf("Error=%d\n", GetLastError());
    return;
}
else {
    printf("\nSuccess on RtAllocateContiguousMemory\n");
    printf("Virtual memory address = 0x%08X\n", vAddress);
    // Get the physical address
    pAddress = RtGetPhysicalAddress(vAddress);
    if (!pAddress.QuadPart) {
        printf("\nFailure on RtGetPhysicalAddress(0x%08X).\t", vAddress);
        printf("Error=%d\n", GetLastError());
    }
    else {
        printf("Success on RtGetPhysicalAddress(0x%08X).\n", vAddress);
    }
    // Free memory
    if (!RtFreeContiguousMemory(vAddress)) {
        printf("\nFailure on
RtFreeContiguousMemory(0x%08X).\t", vAddress);
        printf("Error Code = %d\n", GetLastError());
    }
    else {
        printf("Success on RtFreeContiguousMemory(0x%X).\n", vAddress);
    }
}
}
ExitProcess(0);
}
```



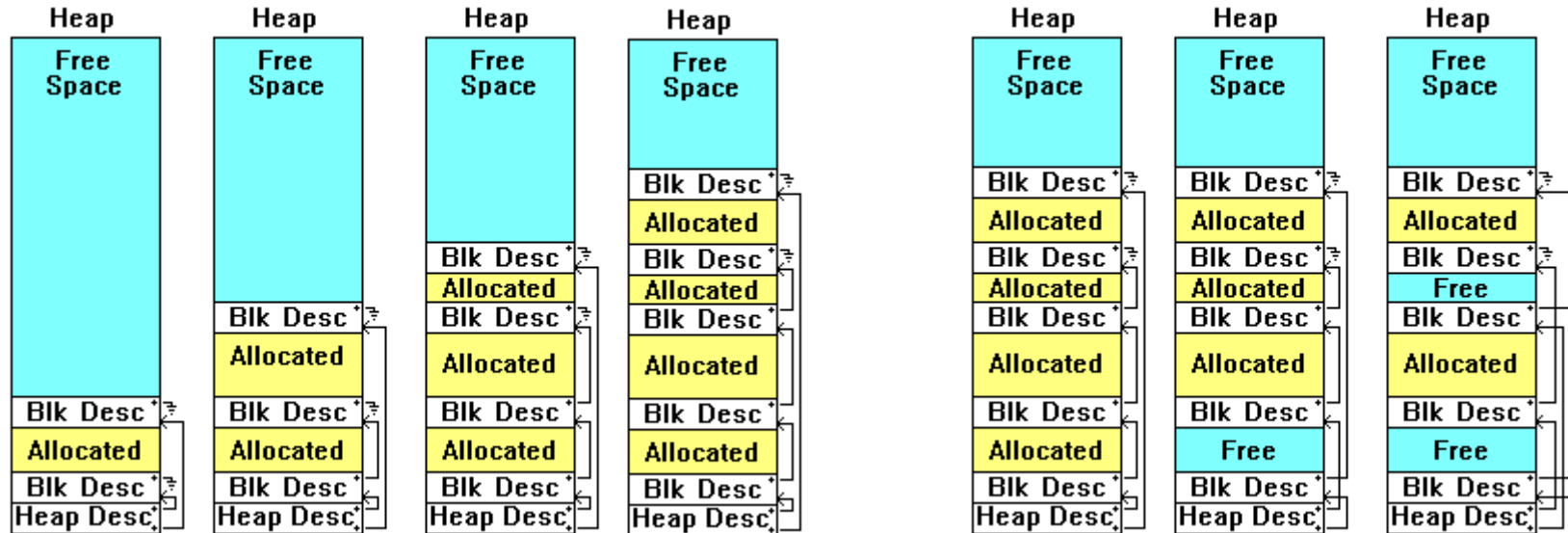
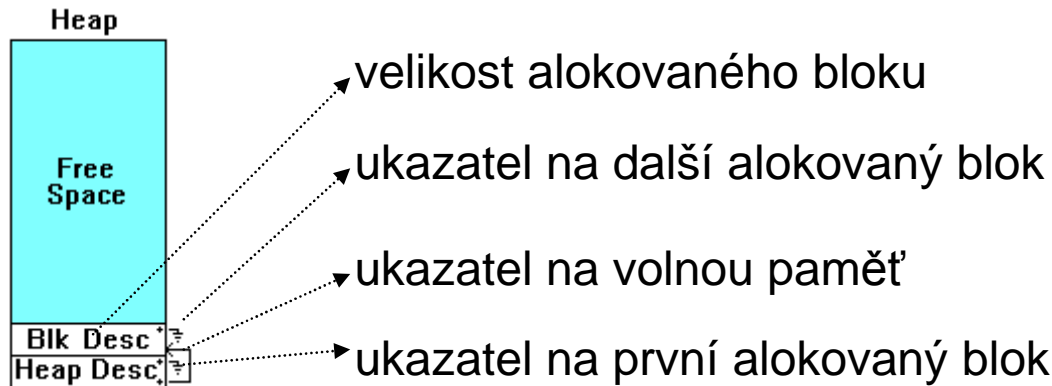
Dynamická alokace

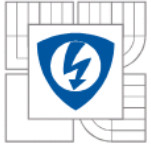
Při dynamické alokaci vzniká problém s velikostí dostupné paměti a deterministickou odezvou při žádosti o paměť. Řešení:

- dynamická alokace paměti při startu (boot phase) – extrémní plýtvání, nemožné v malých RTOS
- dynamická alokace v haldě – problém s fragmentací a real-time odezvou



Heap (halda)





Heap

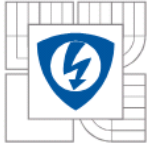
Během chodu aplikace dochází ke ztrátě paměti (v haldě je alokován blok, k němuž neexistuje z aplikace pointer – paměť zůstane neuvolněná). Řešením může být použití smart-pointerů v exception handling.

Alokace je často prováděna po blocích (pool). Bloky by měly mít takovou velikost, aby uspokojily libovolný požadavek o paměť (plýtvání).

Některé RTOS nabízí různé haldy s různě velkými memory-pool.

Částečné řešení je pomocí *garbage collector* mechanismu. V RTOS však velmi problematické. Nutno použít zdvojených pointerů, což zvyšuje výpočetní náročnost jak aplikace, tak čištění paměti.

Mnoho RTOS pak paměť v haldě nenabízí vůbec.



Heap v RTX

RtCommitLockHeap

vymezí pro konkrétní haldu oblast paměti ve fyzické paměti a zabrání jejímu stránkování – WIN32.

GetProcessHeap

vrátí HANDLE na haldu procesu.

HeapAlloc

alokuje blok paměti v haldě. Alokovaná paměť je nepřesunutelná.

HeapReAlloc

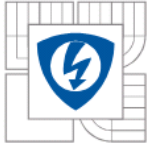
realokuje blok paměti v haldě. Funkce umožní změnit velikost bloku a jeho adresu.

HeapFree

uvolní blok paměti alokované v haldě.

HeapSize

vrátí velikost paměťového bloku alokovaného v haldě.



HeapAlloc

HeapAlloc alokuje blok paměti v hladě. Alokovaná paměť je nepřesunutelná

Prototype

LPVOID

HeapAlloc(

HANDLE *hHeap*,

DWORD *Flags*,

DWORD *Bytes*

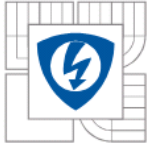
);

Parameters

hHeap **HANDLE** na haldu, ve které bude paměť alokována.

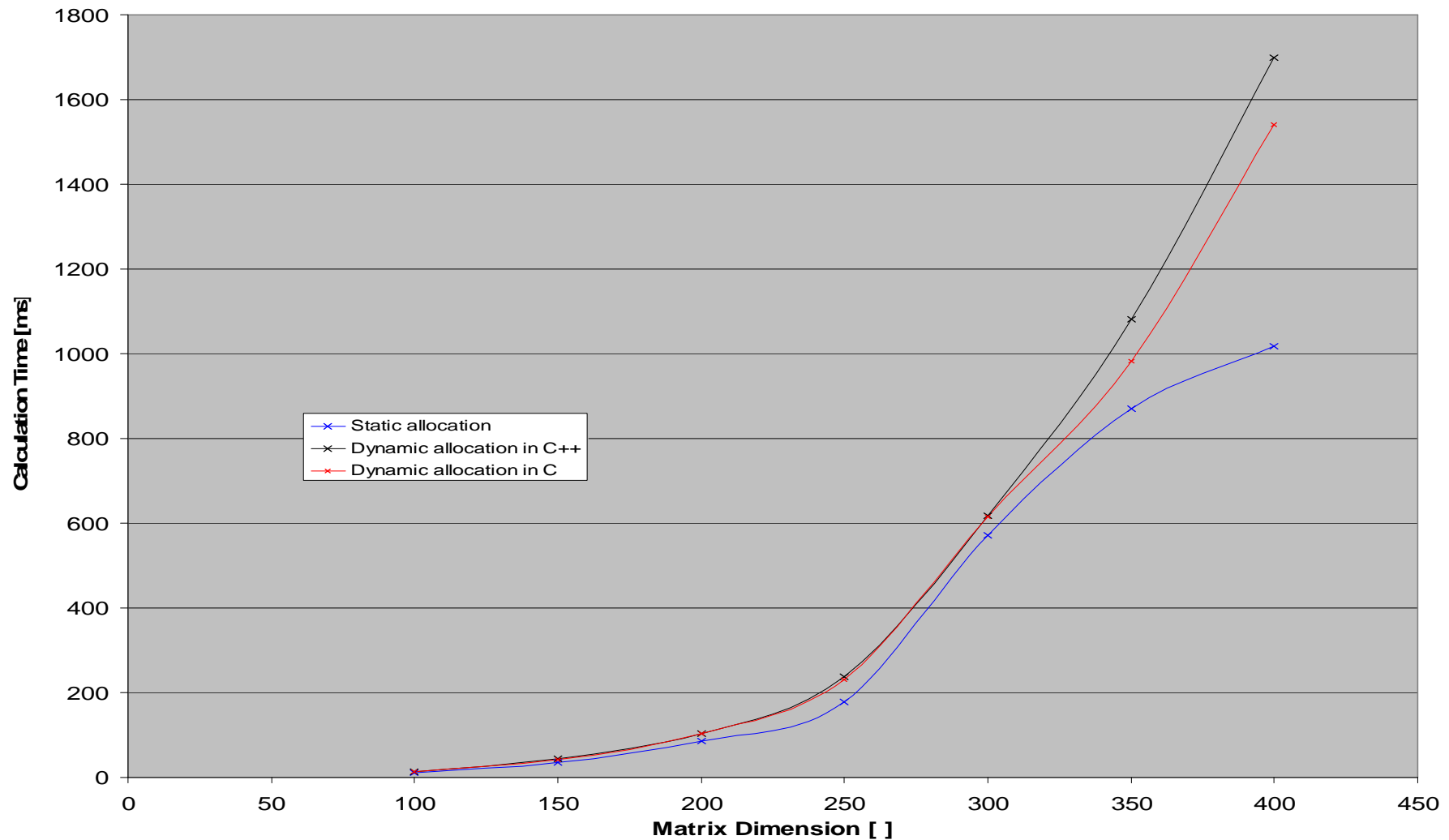
Flags určuje parametry alokovaného bloku. Lze specifikovat:
HEAP_ZERO_MEMORY, kdy bude alokovaná paměť inicializována na nulu.

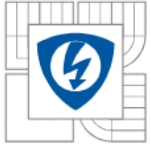
Bytes Velikost alokované paměti v bajtech.



Správa paměti ve WIN32

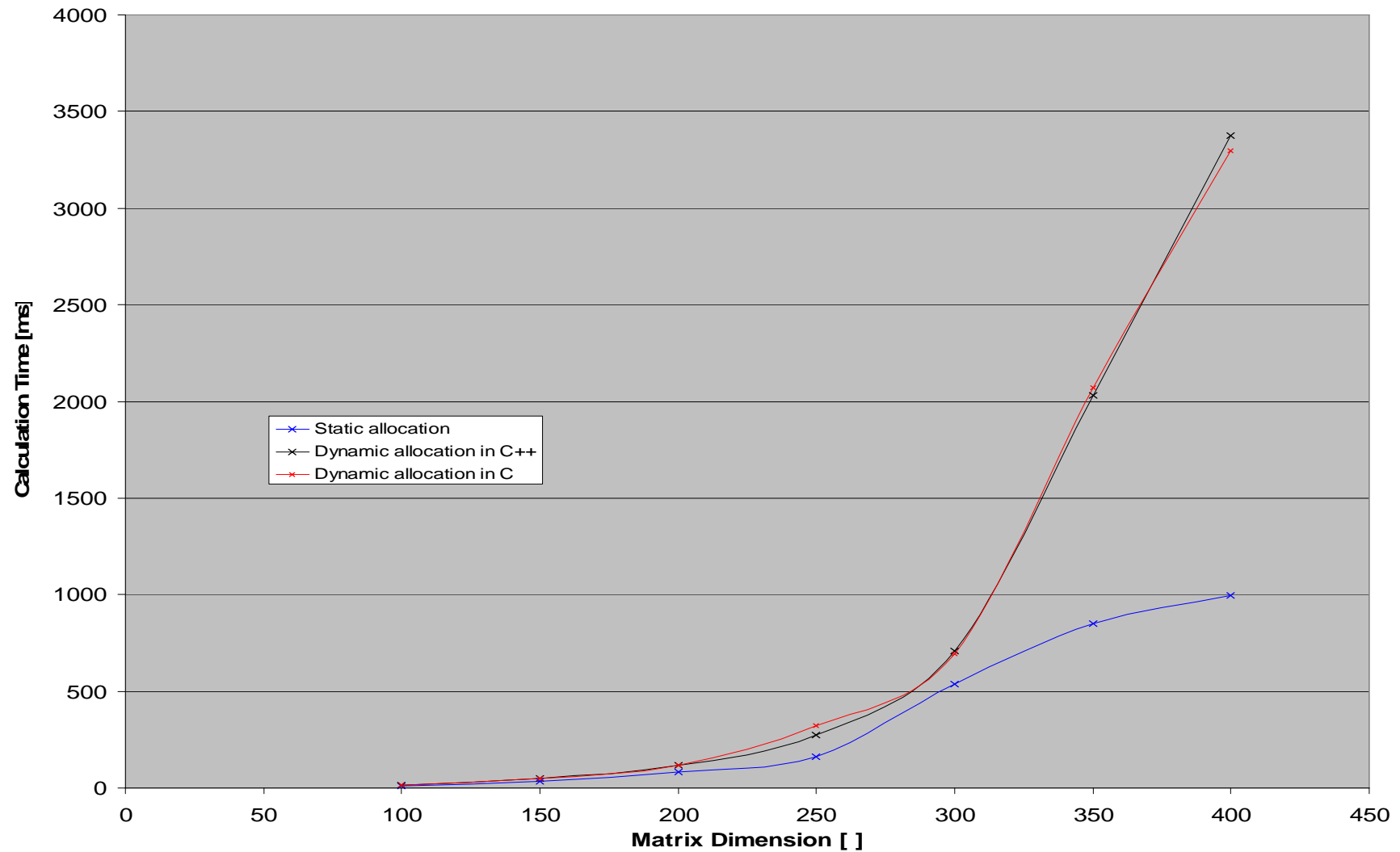
Calculation Time vs. Matrix Dimension in WIN32
AMD 64 X2 Dual Core 4600+, 3GB RAM

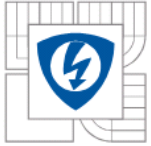




Správa paměti v RTX

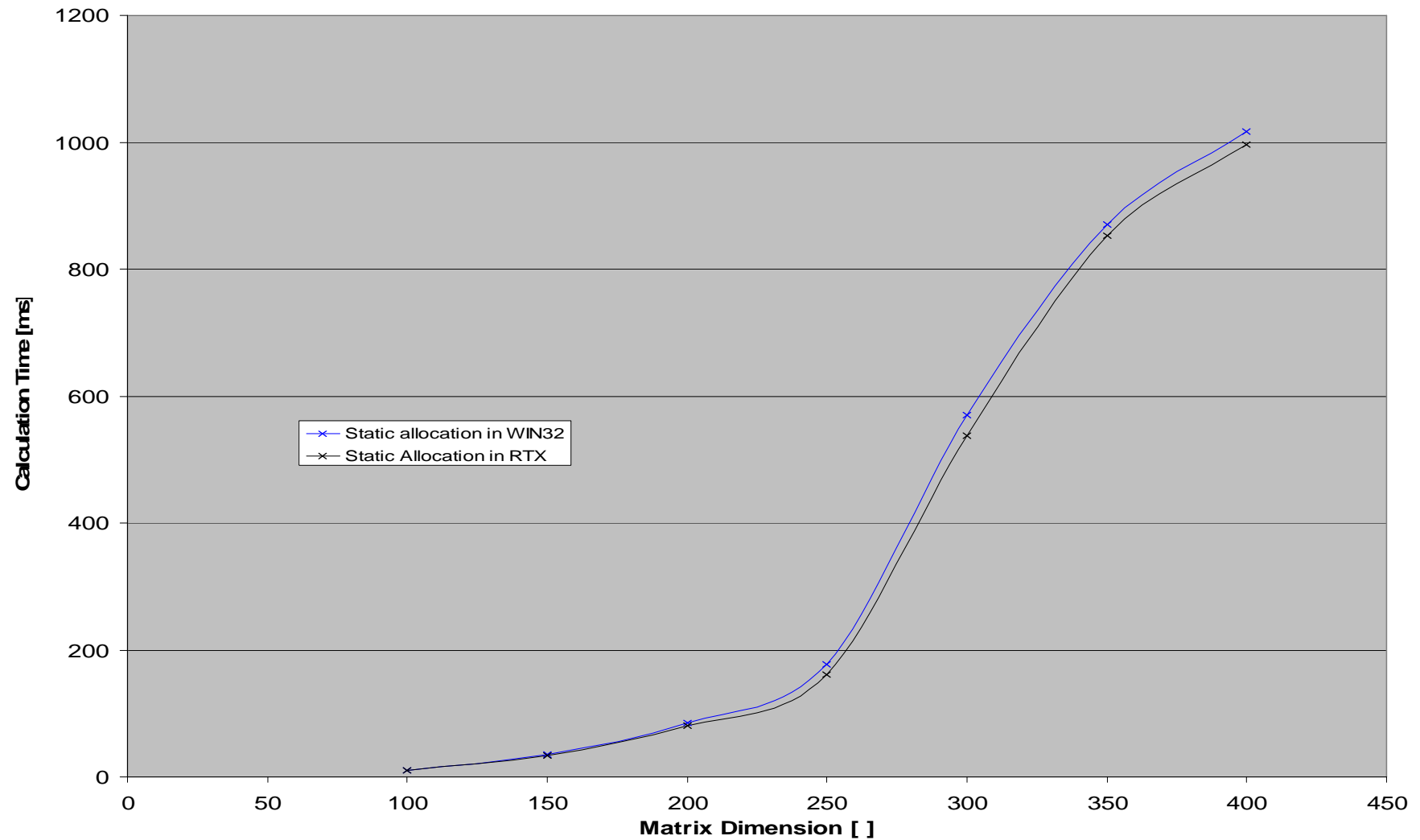
Calculation Time vs. Matrix Dimension in RTX
AMD 64 X2 Dual Core (RTX dedicated) 4600+, 3GB RAM

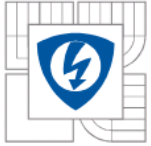




Správa paměti v RTX

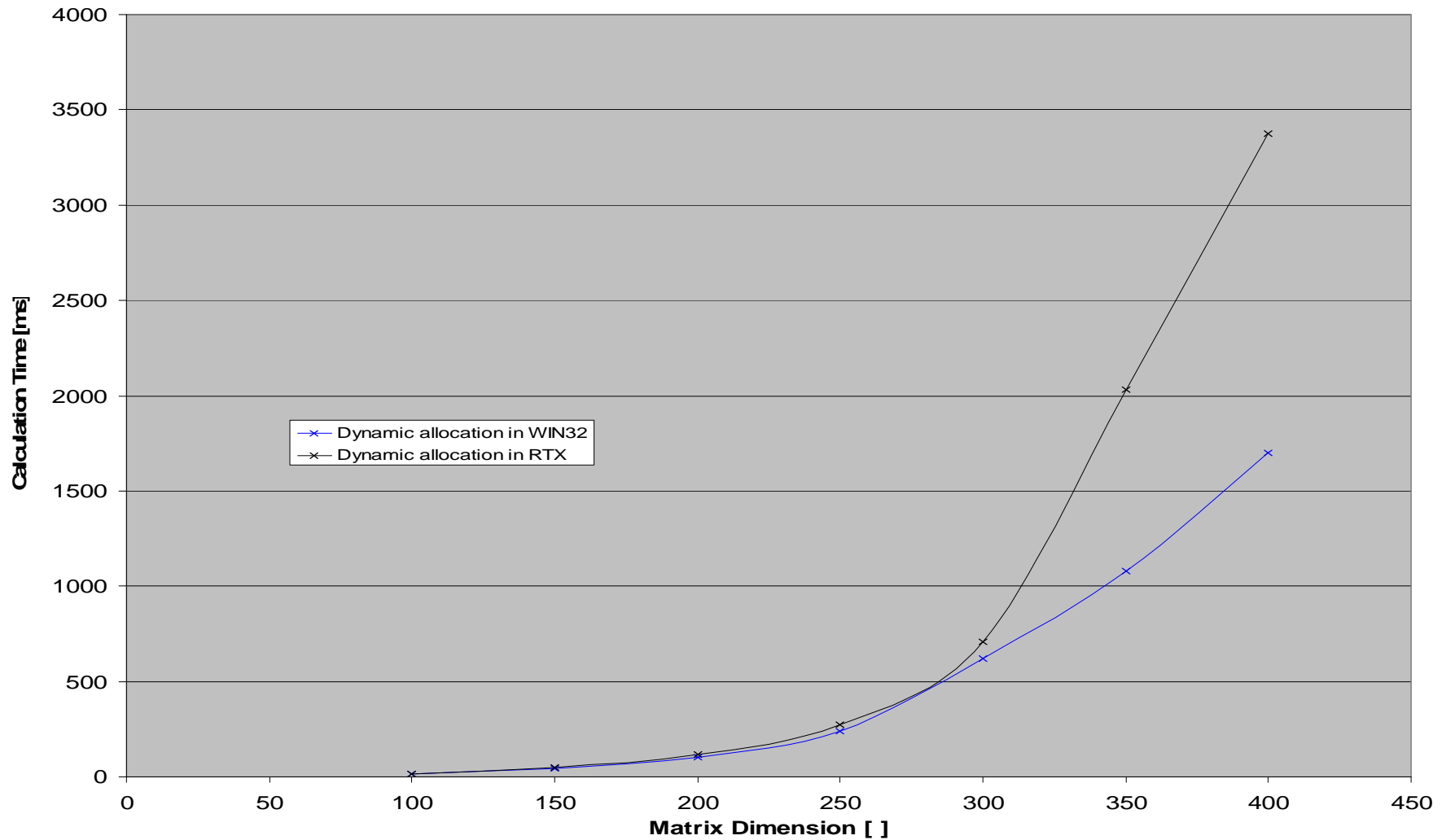
Calculation Time vs. Matrix Dimension
AMD 64 X2 Dual Core 4600+, 3GB RAM

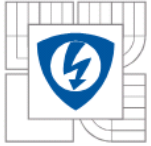




Správa paměti v RTX

Calculation Time vs. Matrix Dimension
AMD 64 X2 Dual Core 4600+, 3GB RAM





Správa paměti v RTX

Calculation Rate vs. Matrix Dimension
AMD 64 X2 Dual Core 4600+, 3GB RAM

