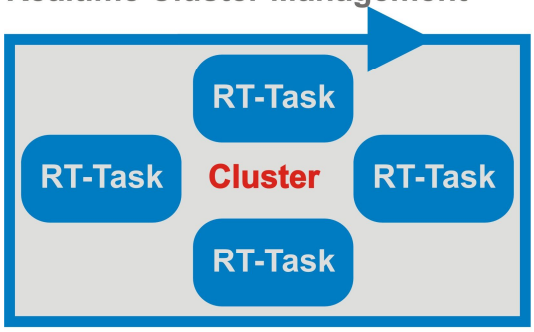
SHA Realtime Engine Documentation (64 Bit)

Date: Oct,26.2022

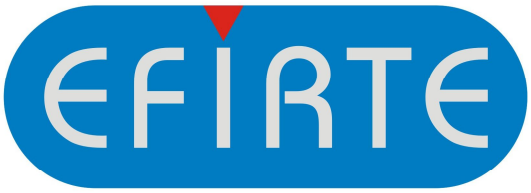
a SYBERA product 

1 Introduction

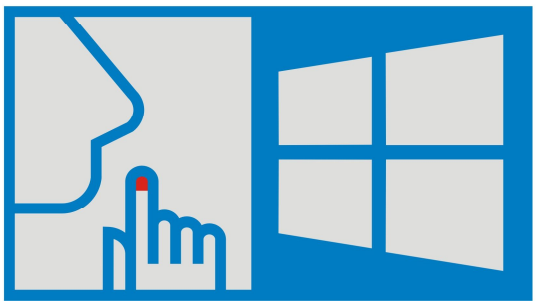
Realtime Cluster Management



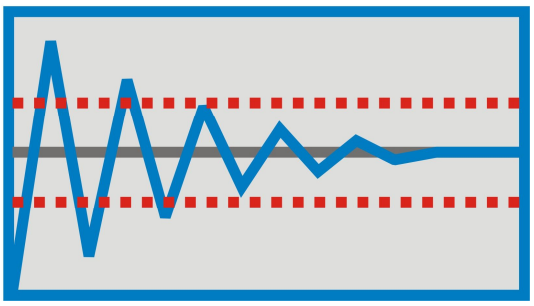
EFI Realtime Engine



Realtime Silent Mode



Adaptive Jitter Reduction



1.1 Realtime programming under Windows 64-bit without compromising

The new Real Time Engine for Windows 64-bit leaves little to be desired. The ability of individual Processors to use as a PLC system - completely decoupled from Windows - guides you into a new realtime dimension. Especially the highly accurate jitter performance allows drive control for SERCOSIII, EtherCAT and Profinet IRT. Each processor can be operated as task-cluster, with single shot or periodic mode and with a frequency up to 200 KHz.

Hardware access under Windows is typically possible only at kernel level. Therefore a realtime task can share usermode and kernelmode. All hardware resources (IO-Space, MappedMemory and DMA) can be easily programmed. A global PCI Enumarator allows you to manage all PCI resources for realtime tasks.

The programming will be done with common IDE interfaces, like VisualStudio - this keep the user interface homogenous. Applications will be designed as true 32 bit solution. If a realtime task raise an exception due to a programming error, the processor may be debugged or rebooted completely independent of the Windows operating system.

1.2 Features:

- Programming of realtime tasks under Windows on a multi core platform
- Native 64 bit mode for application and realtime task
- Boot, Debug and Control of core processors on the same platform
- Building task clusters for core processors
- Free scaling of processors affinity (Windows / Realtime)
- No system crash on processor exception, with easy debugging facilities
- Combination of user-mode code and kernel-mode code within realtime tasks
- Controlling hardware resources within and outside realtime tasks
- High realtime frequency upto 200 KHz, with low jitter
- Dynamic frequency control
- Support of multi-core processors (INTEL ix, AMD Xx, INTEL CoreDuo)
- Very low jitter behaviour by adaptive jitter compensation
- Realtime Silent Mode
- USB Silent Mode
- UEFI BIOS Compatible

1.3 Supported Development Environments

SHA64 was build to support serveral development platforms. Currently following development platforms are supported:

- Visual C++
- Embacadero C++ Builder
- WDK Device Driver Development

1.4 Supported OS

- Windows 10
- Windows 8
- Windows 7

1.5 Supported Platforms

- INTEL Multi-Core (i3 / i5 / i7 / CoreDuo, Atom, XEON)
- AMD Multi-Core (x2 / x4)

1.6 Reference Hardware

SHA64 was specially testet with following hardware platforms:

Mainboard:	DQ57TM
Processor:	INTEL I5
RAM:	4 GB
Graphics:	RADEON 9800 PRO

Mainboard:	ASROCK 785GM-S3
Processor:	ATHLON II X4 320
RAM:	4 GB
Graphics:	ATI Radeon HD 4200 (OnBoard)

Mainboard:	Lenovo 44444XG
Processor:	INTEL Core i5 M480
RAM:	4 GB
Graphics:	INTEL HD Graphics BR-1004 (OnBoard)

1.7 Function Groups

Basic Module	• System functions • Error Logging
Memory Module	• Bus Information • IO Space Access • Mapped Memory Access • Physical DMA Memory
X-Realtime Module	• Processor Control • Processor Information • Task (Cluster) Control • Task (Cluster) Information • Debug Support (inkl. Exception Handling) • Dynamic Frequency Control • Executive Delay • Adaptive Jitter Compensation • Event based Synchronisation

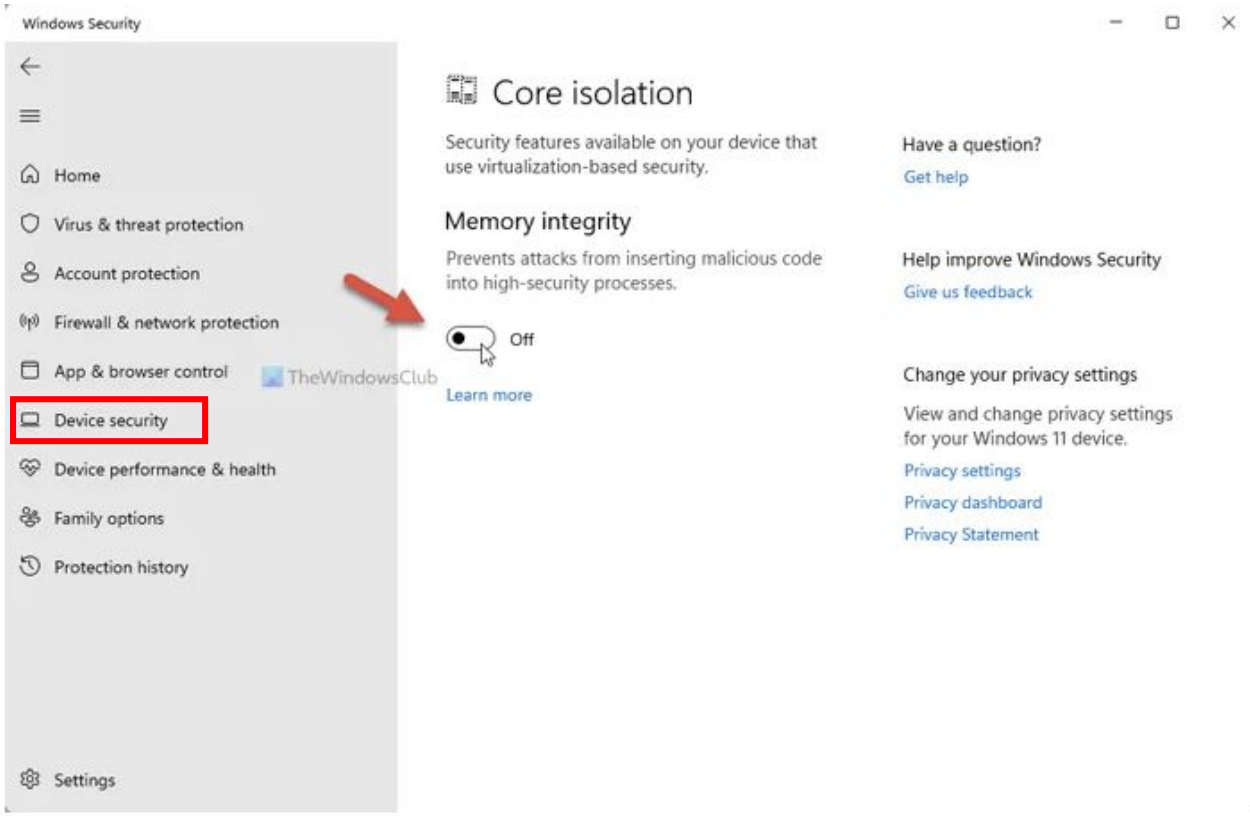
2 BIOS Preparation (recommended)

Do get the most reliable real-time behaviour, its required, to do some BIOS settings:

- Disable Hyper-Threading
- Disable EIST
- Disable Turbo-Mode
- Disable C-States
- Disable USB Legacy Support
- Disable USB OS Hand-Off Synchronization

3 Windows Preparation (required)

The core isolation has to be disabled:



4 Installation

4.1 Check third party software

First make shure that no other realtime software, or subsystem (e.g. BECKHOFF Twincat, NUMEGA SoftICE, ..) is running on the system.

4.2 Preparation

1. Make shure, the latest Updates are installed (especially Windows 7)

2. *Without USB dongle*
   Unpack the SHA64xxx.ZIP file on the hard disk
   (make sure the directory path has no space characters)

   With USB dongle
   Just start installation from USB dongle

4.3 Installation

1. Run SYSETUP64.EXE (as Administrator)
2. Reboot the system
3. Optional: Check license with SYLICENCECHECK64.EXE

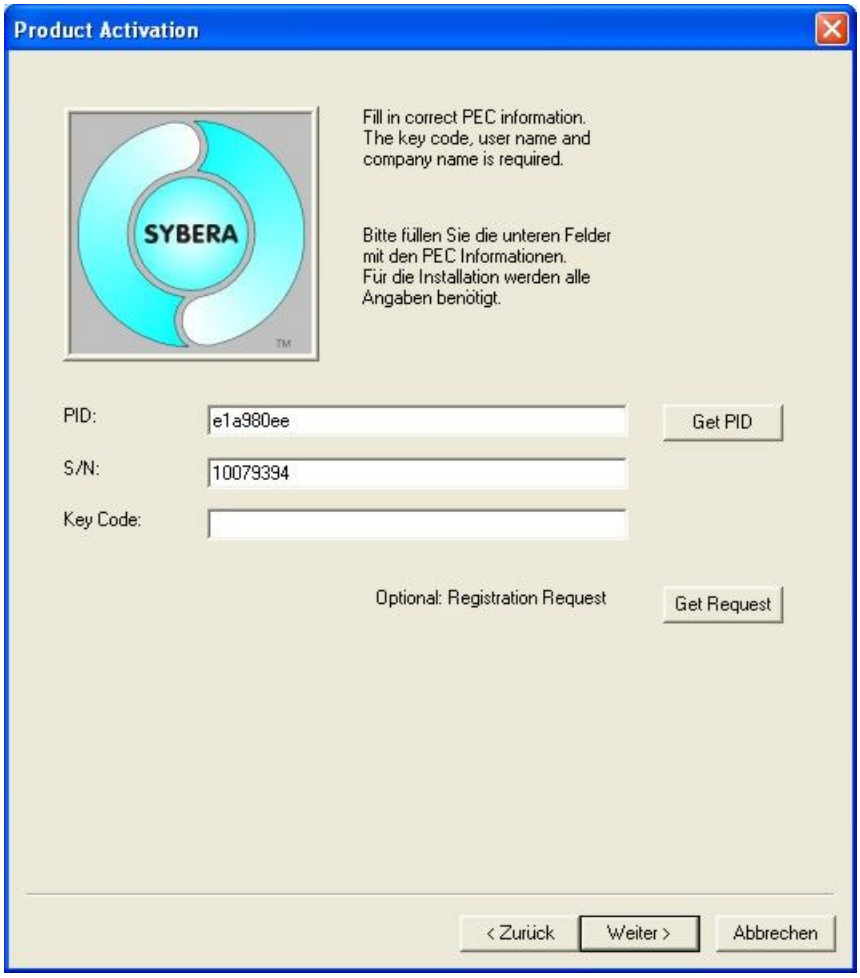
4.3.1 Product Activation

For product activation a PEC-Code (Product Enabling Code) is required, consisting of 3 items:

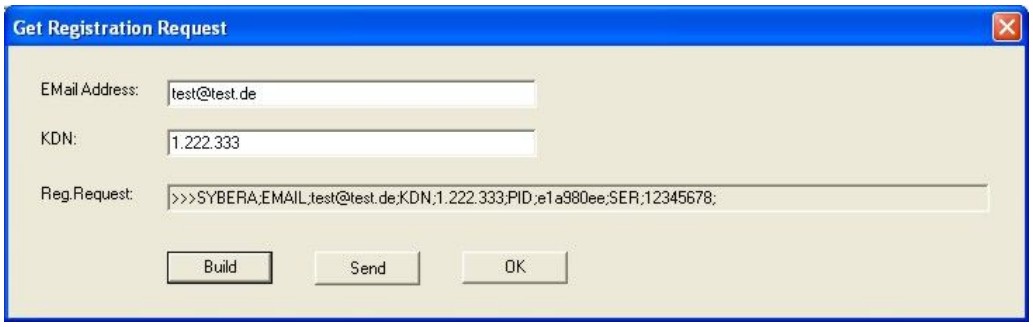
- PID: Platform Identifier (received by pressing the button)
- S/N: Serial number (received from license label or 12345678 for evaluation)
- KeyCode: Installation Code (received from license server or 00001111-22223333 for evaluation)

Each License is bound to a single PC-Platform by a corresponding PID-Code. This PID-Code is to be read out at Installation with the SYSETUP-Software, and sent to the SYBERA registration server for generating the valid PEC-Code (not required for Open-Volume-Licenses).

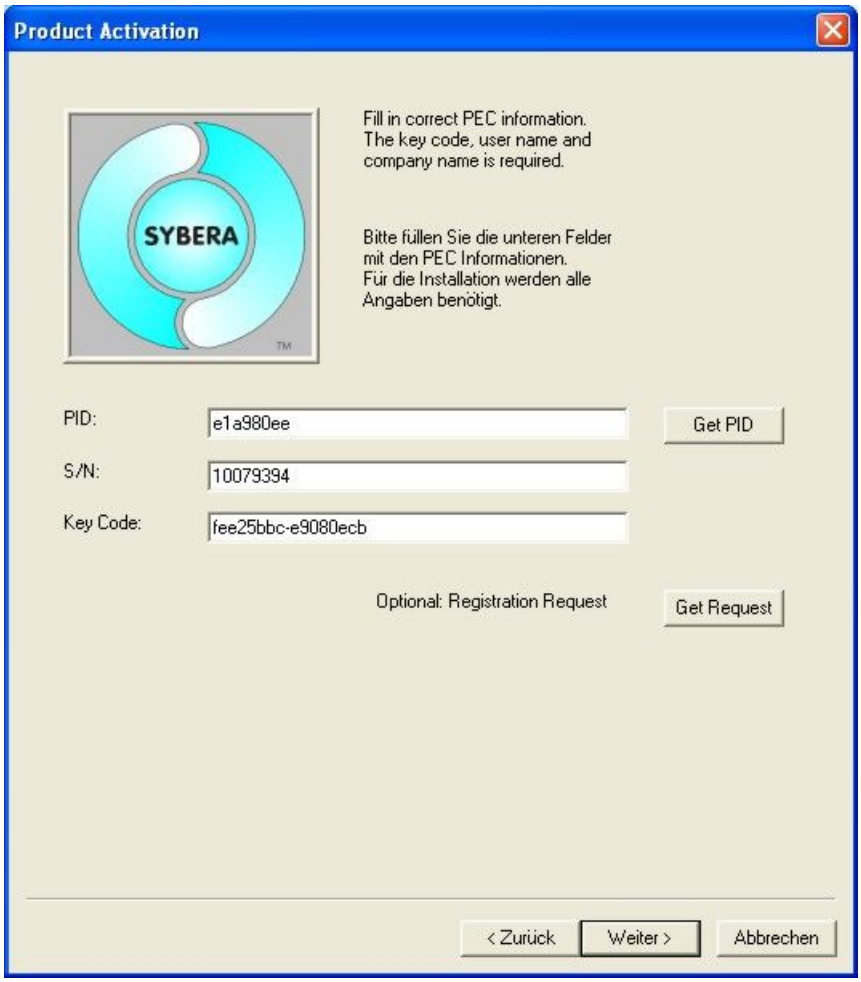
Get the PID code and enter the serial number



Depending if the PC is online, the KeyCode may be requested direct or indirect from the SYBERA registration server (see the registration manual)

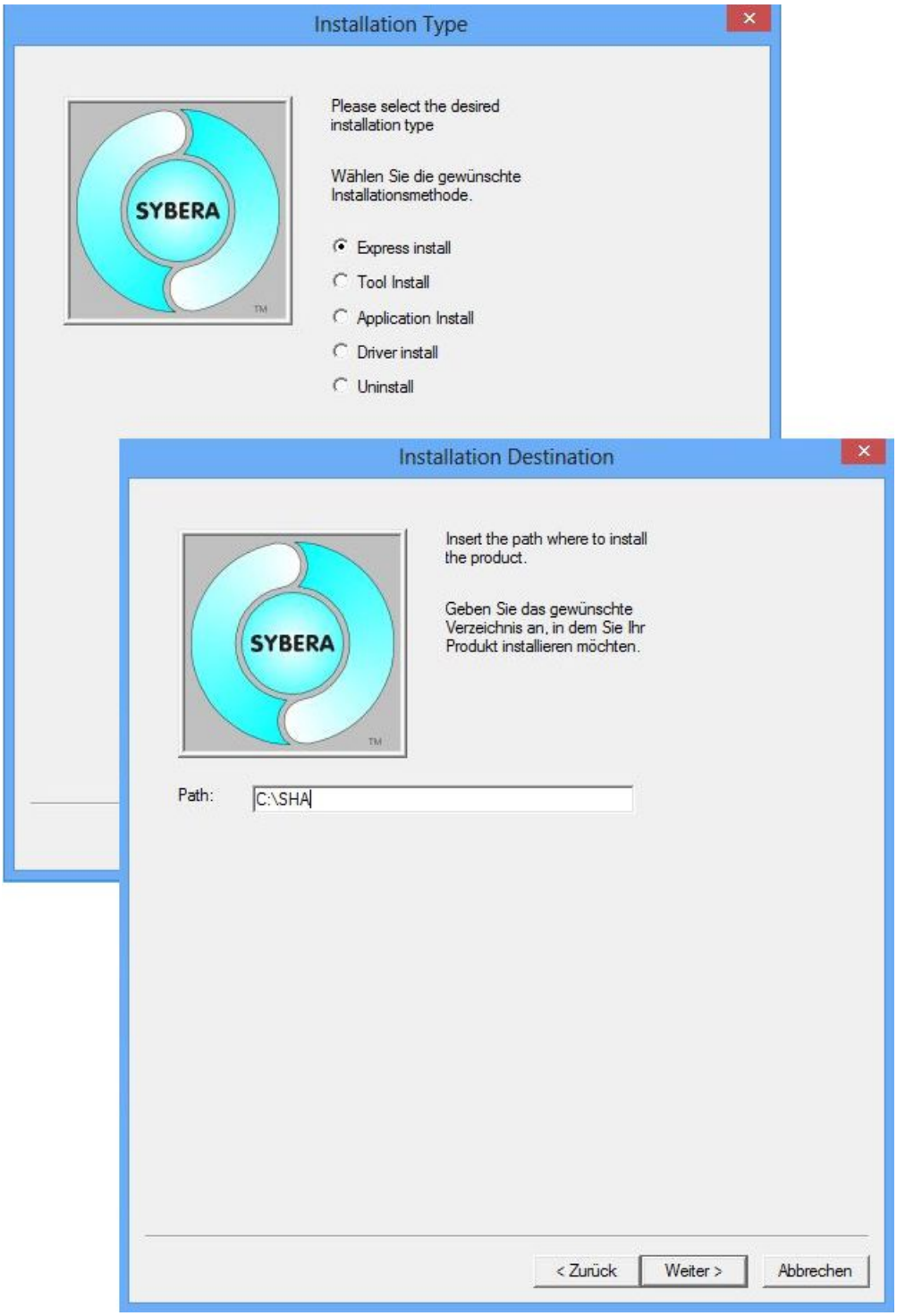


The SYBERA registration server will now provide the required PEC information for installation



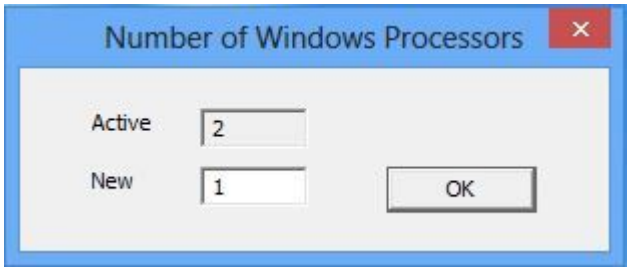
4.3.2 Installation Type

Typically choose Express Install. All nessecary files will be copied to the destination path.



4.3.3 Number of Processors for Windows

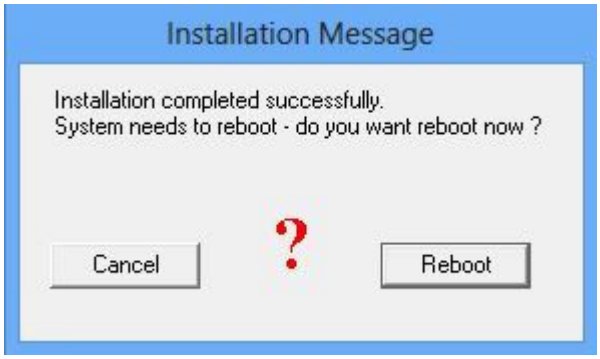
Choose the number of processors (**to be used by windows)**.



Note: The best Jitter behaviour is achieved by setting 1 Processor for Windows. This is due to internal communication between the processors.

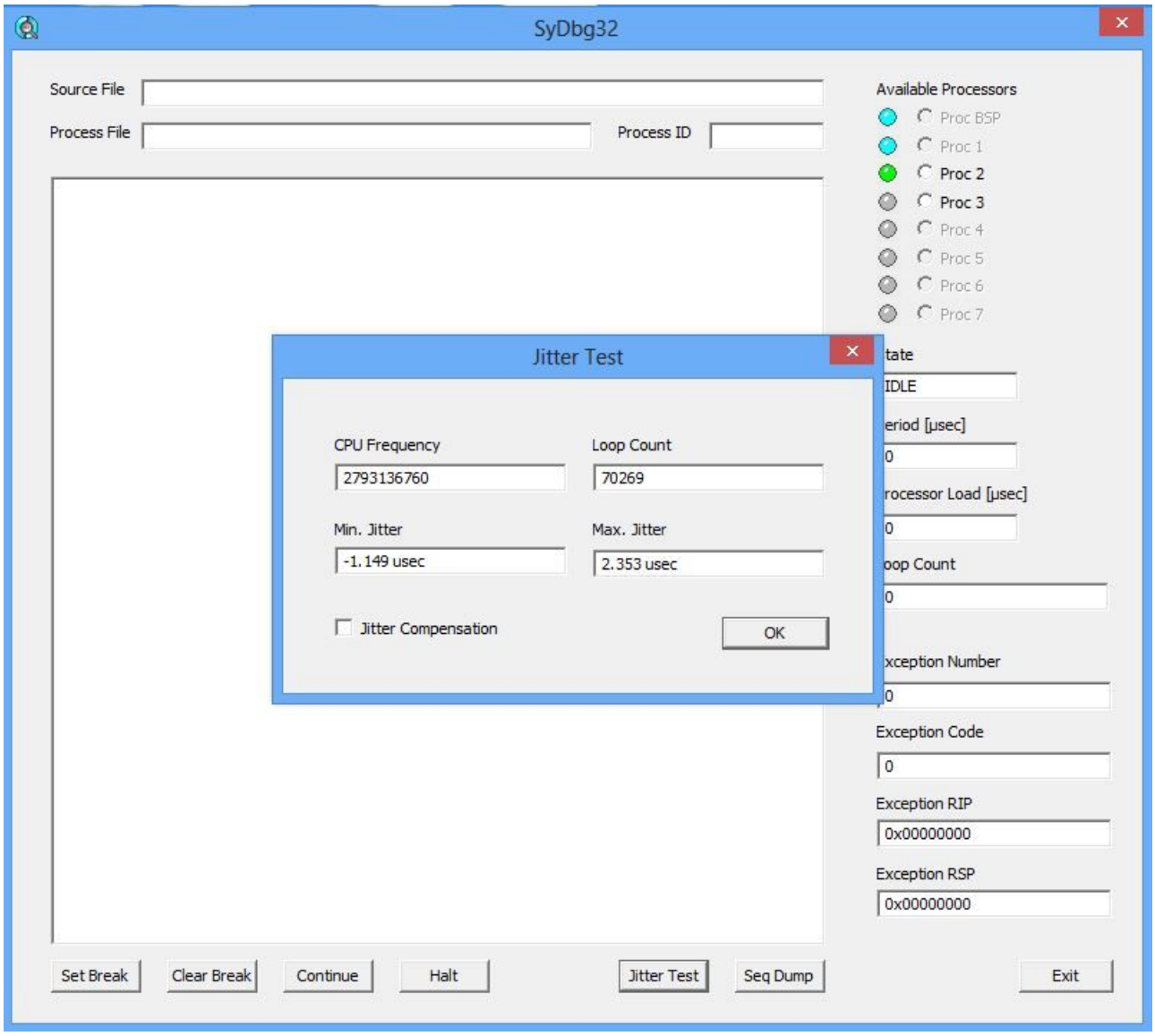
4.3.4 Complete Installation

After proceeding all installation steps a reboot is required



4.3.5 Test the Realtime Engine

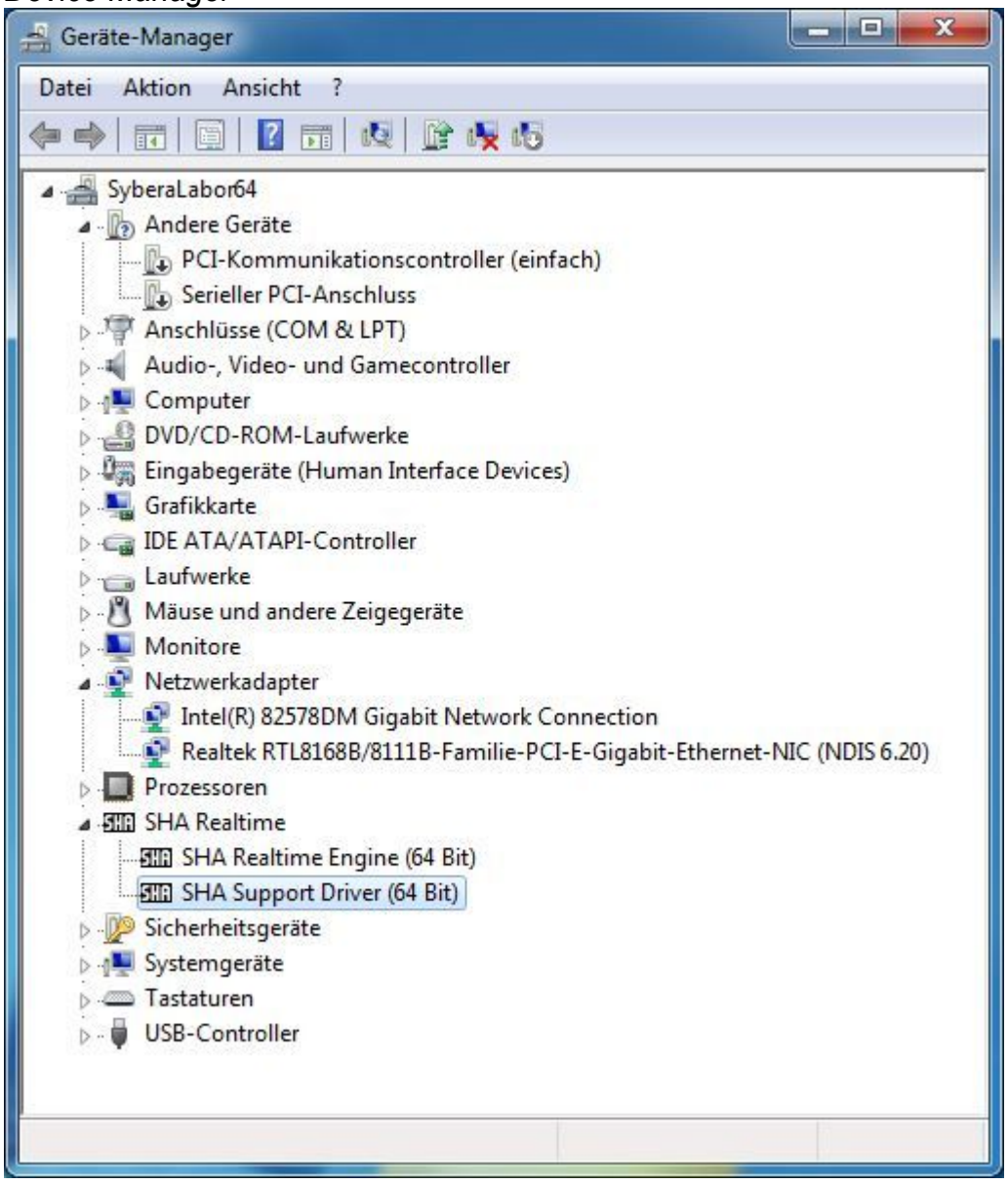
SYBERA provides a program called SYDBG64 which checks the system plattform and tests the realtime engine. On demand it claims a free processor core for a jitter test. The SYDBG program may also be used to debug already running realtime applications and control the selected processor core. The program is found in the directory [\App]



4.4 Installing the WDM support driver for a hardware device (optional)

When a new device is plugged in, the Hardware-Wizard appears and requires a corresponding WDM device driver for this hardware. The required driver is SHA64SUPP within the directory [\SUPPORT]. Do not install the driver SHA64DRV.SYS, nor any corresponding vendor device driver. Please follow chapter *PNP resource management* for changing the INF file.

Device Manager



The SHA Support Driver has to be installed at PNP device enumeration of Windows. There are two types of hardware requiring a PNP Support Driver:

a) a system known PNP Hardware was added (e.g. ethernet adapter)
b) a system unknown PNP Hardware was added (e.g. industrial IO adapter)

In the first case Windows automatically installs its own PNP driver for the known hardware and claims therefor all found hardware resources. To access these resources and avoiding any resource conflicts when using SHA, the origin driver must be ***replaced*** by the SHA support driver. The SHA support driver may be included several times, if more than one PNP devices are installed.

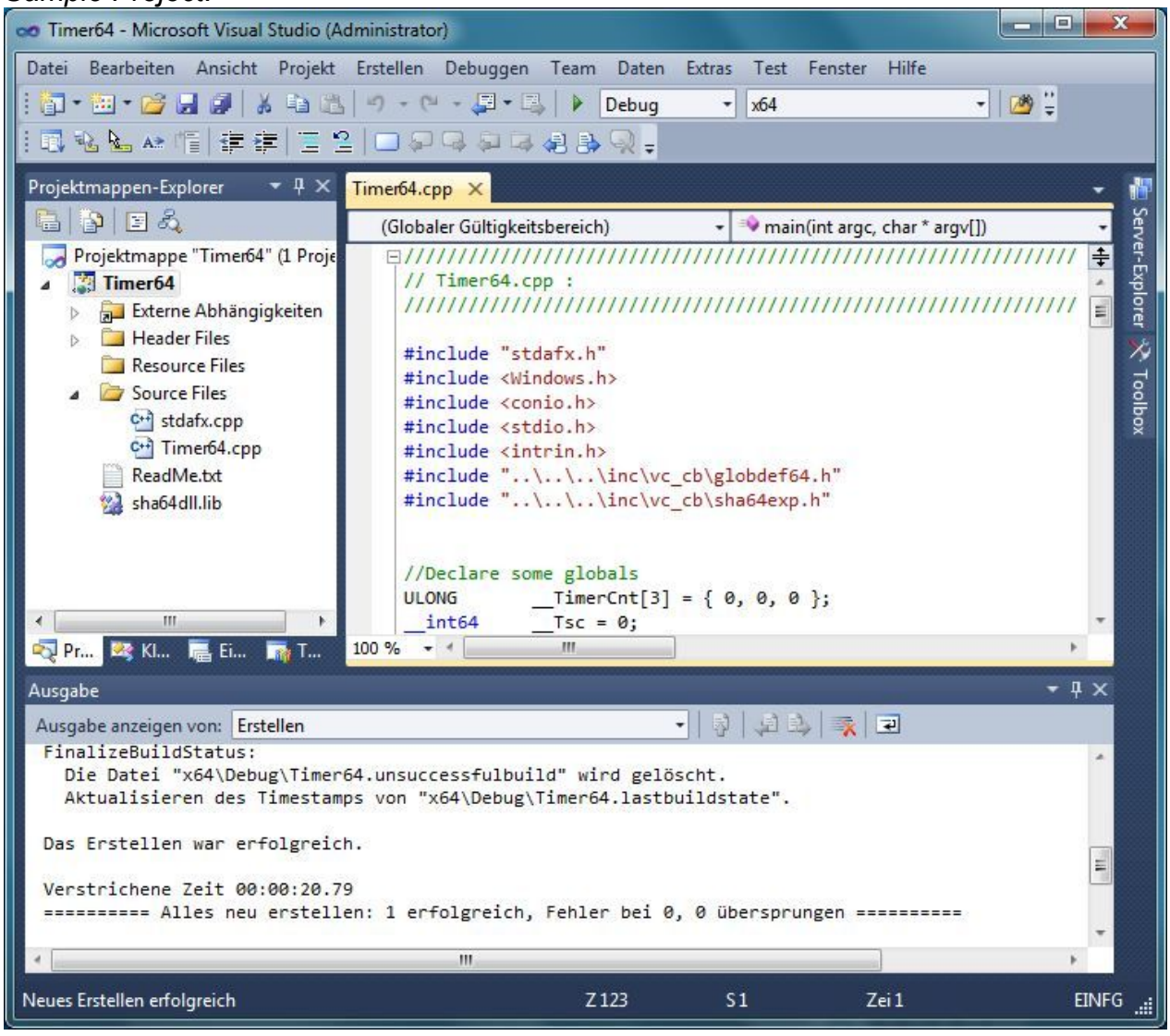
In the second case Windows will ask for a separate PNP device driver. On installation ignore any setup warnings, since these depend on several device groups.

5 SHA64 Libraries

SHA exports its functions by several libraries. Following header and libraries files are shipped:

```
LIB\SHA64DLL.LIB              Project Import Library
LIB\SHA64DLL.DLL              Dynamic Link Library (copied to \WINDOWS\SYSTEM32)
LIB\SHA64IOPORT.OBJ           IO-Port function library
LIB\SHA64TIME.OBJ             Time function library
INC\GLOBDEF64.H               Global Definitions
INC\SHA64DEBUG.H              Macros for USB Silent Mode
INC\SHA64MACROS.H             Macro Definitions
INC\SHA64EXP.H                Exported Function Prototypes
INC\SHA64SYNC.H               Synchronisation Macros
INC\SHA64USB.H                USB Silent Mode
```

Sample Project:



6 Basic Module

The basic module provides several functions for system information and management. Following functions are available:

6.1 Sha64GetVersion

This function allows to evaluate the current release version of the driver and DLL. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64GetVersion(
                  double* pdDllVersion,   //OUT: DLL Version
                  double* pdDrvVersion)   //OUT: DRV Version
```

6.2 Sha64GetSystemInfo

This function gets internal system information, mainly as parameters for other functions. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

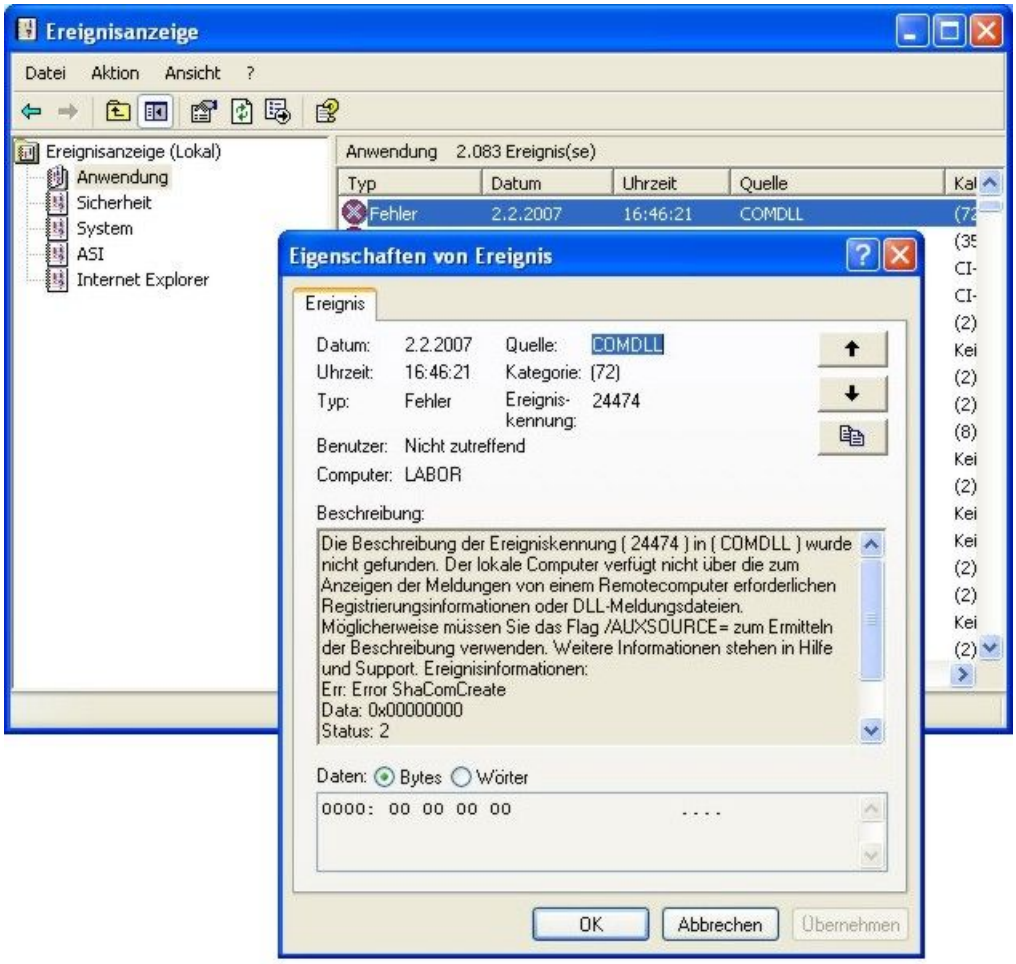
```
ULONG Sha64GetSystemInfo(
                  PULONG pWinProcNum,     //OUT: Number of windows processors
                  PULONG pPhyProcNum,     //OUT: Number of physical processors
                  PULONG pLogProcNum)     //OUT: Number of logical processors
```

6.3 Sha64LogError

This function logs an error to the system event log. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64LogError(
                  char* pszSrc,     //IN: Name of source module
                  char* pszFile,    //IN: Name of source file
                  char* pszFunc,    //IN: Name of function
                  char* pszErr,     //IN: Error message
                  ULONG SrcLine,    //IN: Source line number
                  ULONG ErrCode,    //IN: Error code
                  ULONG ErrData,    //IN: Error data
                  ULONG Status)     //IN: Status
```

EventViewer



For error logging following predefined MACROS are predefined:

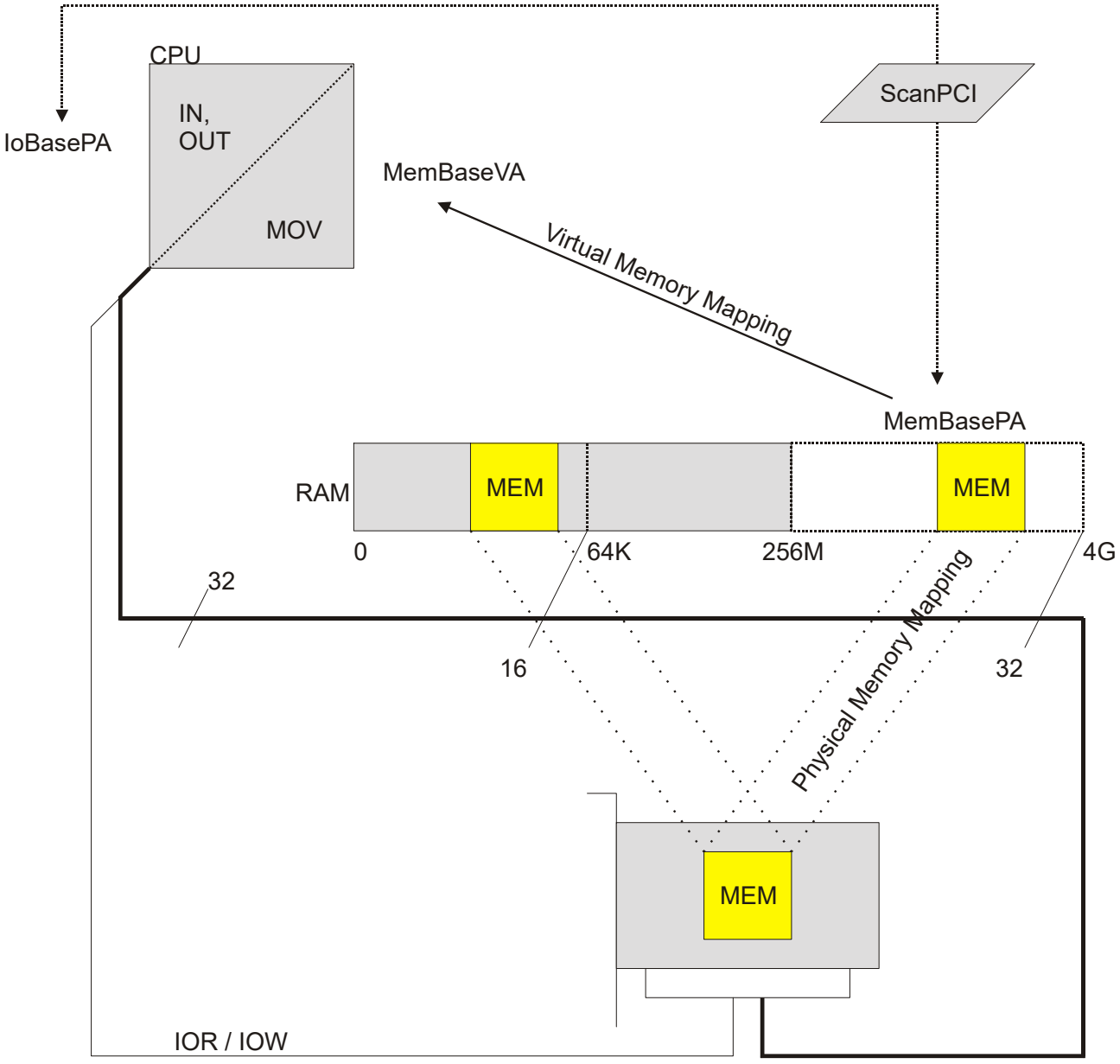
```
#define ERROR_LOG(__Src, __Status) \
Sha64LogError(__Src,__FILENAME__,__FUNCTION__,"Unknown",__LINE__,0,0,__Status)

#define ERROR_LOG_MSG(__Src, __Msg, __Status)\
Sha64LogError(__Src,__FILENAME__,__FUNCTION__,__Msg,__LINE__,0,0,__Status)

#define ERROR_LOG_VAL(__Src, __Val, __Status)\
Sha64LogError(__Src,__FILENAME__,__FUNCTION__,"Unknown",__LINE__,0,__Val,__Status)
```

Sample:

```
//Try to open device link
*phDrv = CreateFile(
                  szServiceName,
                  GENERIC_WRITE | GENERIC_READ,
                  0, NULL,
                  OPEN_EXISTING,
                  FILE_ATTRIBUTE_NORMAL,
                  NULL);

if (*phDrv == INVALID_HANDLE_VALUE)
      return ERROR_LOG("MYMODULE", GetLastError());
```

7 Memory Module

This module allows IO port access, physical mapped memory access, reading and modifying values within the PCI configuration space. IO-Port accesses are the most common hardware operations, because of direct use of physical addresses by separeted CPU commands. With mapped memory, a physical storage area of the adapter is connected to the system storage and is treated as system memory. To gain programming access a virtual address region has to be mapped to the physical memory.



7.1 Sha64GetDevInfo

This function receives resource information about PNP devices. It allows a filtered enumeration of all PNP devices within the system. Typically its used to get information about PCI bus devices due to their vendor and device IDs. On success the returning value is ERROR_SUCCESS and the datablock contains all necessary information about the found PCI device, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64GetDevInfo(
                  WCHAR* wszFilterHardwareID,   //IN: Hardware ID
                  ULONG FilterInstance,         //IN: Instance number
                  ULONG DataType,               //IN: BUS data type
                  ULONG DataOffs,               //IN: Data offset
                  ULONG DataSize,               //IN: Data size
                  PVOID pDataBlock,             //OUT: pointer to Data block
                  PULONG pBusNumber,            //OUT: bus number (optional)
                  PULONG pAddress,              //OUT: Device address (optional)
                  PULONG pNodeStatus,           //OUT: Node status (optional)
                  WCHAR* wszDevDesc)            //OUT: Device description
```

Note: All required constants are defined within the header file GLOBDEF32.H

Sample: Enumeration of all PCI devices

```
ULONG Error;
WCHAR wszDevDesc[MAX_PATH] = { 0 };
PCI_HEADER PciHeader;
ULONG BusNumber;
ULONG NodeStatus;


ULONG i=0;
do
{
      Error = Sha64GetDevInfo(
                        L"PCI",                         //Hardware ID
                        i,                              //Instance
                        PCI_WHICHSPACE_CONFIG,          //Data Type
                        0,                              //Data Offset
                        sizeof(PCI_HEADER),             //Data Size
                        &PciHeader,                     //Data Buffer
                        &BusNumber,                     //Bus Number (optional)
                        NULL,                           //Address (optional)
                        NULL,                            //Node Status (optional)
                        wszDevDesc);                    //Device Description
      i++;
}
while (wszDevDesc[0] != 0);
```

7.2 Sha64ReadPort

Reading of a value from the IO port. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64ReadPort(
                  USHORT Port,      //IN  : Physical port address
                  ULONG  Size,      //IN  : Port size (1,2 or 4 Byte)
                  PVOID  pValue)    //OUT : Port value
```

7.3 Sha64WritePort

Writing a value to the IO port. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64WritePort(
                  USHORT Port,      //IN  : Physical port address
                  ULONG  Size,      //IN  : Port size (1,2 or 4 Byte)
                  ULONG  Value)     //IN  : Port value
```

Sample:

```
DWORD Error = ShaWritePort(0xE000, 1, (ULONG)0x55);
```

7.4 Sha64SytemReadPortxxx

Reading of a value from the IO port within a cluster task. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
//IN: Physical port address, IN: Value pointer
ULONG Sha64ReadPortUchar( USHORT Port, PUCHAR pVALUE)
ULONG Sha64ReadPortUshort(USHORT Port, PUSHORT pVALUE)
ULONG Sha64ReadPortUlong( USHORT Port, PULONG pVALUE)
```

7.5 Sha64SytemWritePortxxx

Reading of a value from the IO port within a cluster task. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
//IN: Physical port address, IN: Value
ULONG Sha64WritePortUchar( USHORT Port, UCHAR VALUE)
ULONG Sha64WritePortUshort(USHORT Port, USHORT VALUE)
ULONG Sha64WritePortUlong( USHORT Port, ULONG VALUE)
```

7.6 Sha64MapMem

This routine maps a virtual address region to the physical memory area. On success the returning value becomes ERROR_SUCCESS and the pointer(s) to the address space becomes valid for the mapped memory area, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64MapMem(
            ULONG MemPA,            //IN  : Physical memory address
            ULONG MemSize,          //IN  : Memory size
            PVOID* ppMemUserVA,     //OUT : Pointer for windows task access
            PVOID* ppMemSystemVA,   //OUT : Pointer for realtime task access
            HANDLE* phMapmem)       //OUT : Handle to memory device
```

7.7 Sha64UnmapMem

Release of the mapped memory area. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

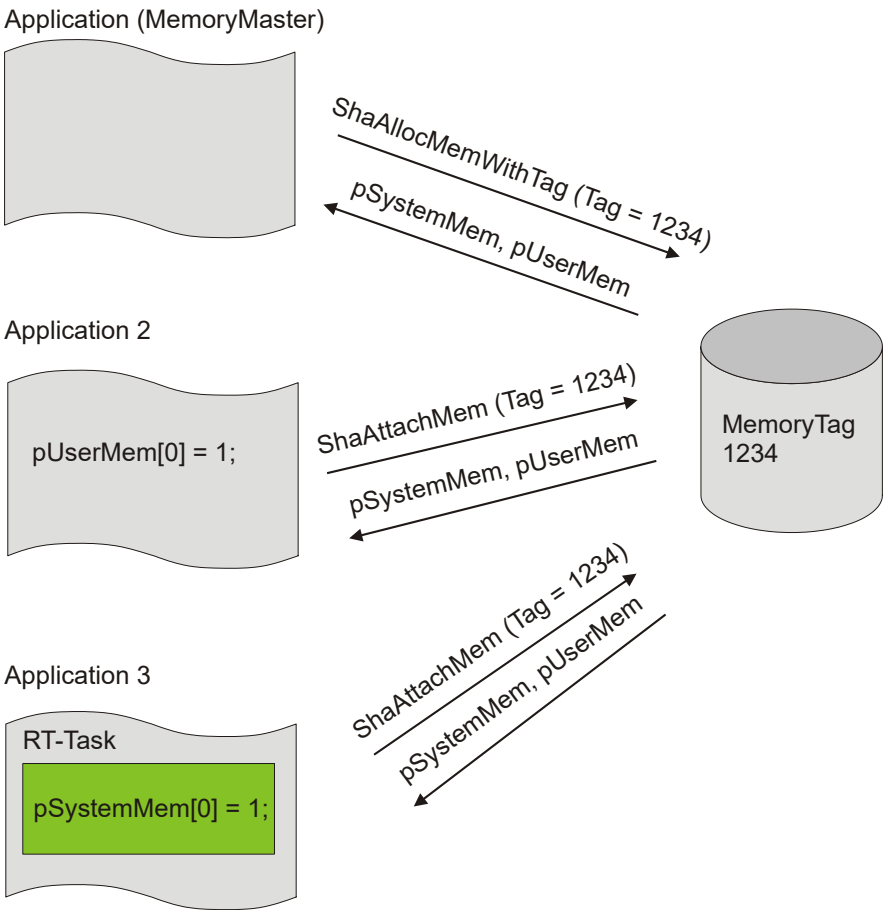
```
ULONG Sha64UnmapMem(HANDLE hMem)    //IN : Handle to memory device
```

Sample:

```
//Declare some globals
PULONG __pMemUserVA = NULL;
PULONG __pMemSystemVA = NULL;

void ProcessorTask(void)
{
      if (__pMemSystemVA)
          __pMemSystemVA[2]++;
}

int main(int argc, char* argv[])
{
      ULONG Error = 0;
      HANDLE hMapmem = NULL;
      PCI_HEADER PciHeader;
      ULONG BusNumber;
      WCHAR wszDevDesc[MAX_PATH] = { 0 };

      Error = Sha64GetDevInfo(
                        L"PCI\\VEN_10EC&DEV_8169",    //Hardware ID
                        0,                            //Instance
                        PCI_WHICHSPACE_CONFIG,        //Data Type
                        0,                            //Data Offset
                        sizeof(PCI_HEADER),           //Data Size
                        &PciHeader,                   //Data Buffer
                        &BusNumber,                   //Bus Number (optional)
                        NULL,                         //Address (optional)
                        NULL,                         //Node Status (optional)
                        wszDevDesc);                  //Device Description

      //Map memory
      Error = Sha64MapMem(
                        PciHeader.u.type0.BaseAddresses[1],
                        0x80,
                        (PVOID*)&__pMemUserVA,
                        (PVOID*)&__pMemSystemVA,
                        &hMapmem);

      __pMemUserVA[2] = 0x11223344;

      //Cleanup
      Sha64UnmapMem(hMapmem);
      return 0;
}
```

7.8 Sha64AllocMemWithTag

Allocation and reserving of physical contiguous memory. On success the returning value is ERROR_SUCCESS and the pointer „pUserVA“ becomes valid for Windows function access and „pSystemVA“ for realtime task access, as well as the physical address. Since the memory is allocated from a limited system resource, it’s not garanteed to obtain the memory (typically memory sizes greater than 4 MB).



```
ULONG Sha64AllocMemWithTag (
            BOOLEAN CachedFlag, //IN  : Allows cached physical memory
            ULONG Tag,          //IN  : (Optional) Memory TAG
            long* pSize,        //IN  : Memory size
            PVOID* ppUserVA,    //OUT : Pointer to memory in Windows function
            PVOID* ppSystemVA,  //OUT : Pointer to memory in Realtime task
            ULONG* pPA,         //OUT : Physical memory address
            HANDLE* phMem)      //OUT : Handle to memory device
```

Note:
Whether the demanded storage can be reserved or not depends on the already used system resources and the fragmented virtual memory. As a rule up to approx. 4 MB of contiguous physical memory can be allocated without problem.

7.9 Sha64FreeMem

Release the allocated physical memory. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64FreeMem(HANDLE hMem)          //IN : Handle to memory device
```

Sample:

```
//Declare some globals
PULONG      __pUserMem = NULL;
PULONG      __pSystemMem = NULL;
HANDLE      __hMem = NULL;


void TimerTask(void)
{
      //Update memory
      if (__hMem) { __pSystemMem[0]++; }
}

int main(int argc, char* argv[])
{
      ULONG Error = 0;
      HANDLE hProc = NULL;
      HANDLE hTask = NULL;
      ULONG MemBasePA;
      ULONG MemSize = 0x100000;

      //Allocate memory
      if (ERROR_SUCCESS == Sha64AllocMemWithTag(
                        FALSE,                  //Cached memory
                        0x1234,                 //Memory tag
                        MemSize,                //Memory size
                        (void**)&__pUserMem,    //Pointer for Windows function
                        (void**)&__pSystemMem,  //Pointer for Realtime task
                        &MemBasePA,             //Physical memory address
                        &__hMem))               //Handle to memory device
      {
            //Init physical memory
            memset(__pUserMem, 0, MemSize);

            Error = Sha64CreateProcessor(REALTIME_PERIOD, FALSE, NULL, &hProc);
            Error = Sha64CreateTask(TimerTask, 1, 1000, FALSE, NULL, hProc, &hTask);
            Error = Sha64ControlProcessor(hProc, PROC_CTRL_CONTINUE, 0);

            //Wait for key press
            while (!_kbhit())
            {
                  printf("Press any key ... [%08x]\r", __pUserMem[0]);
                  Sleep(500);
            }
            //Cleanup
            Sha64DestroyTask(hTask);
            Sha64DestroyProcessor(hProc);
            Sha64FreeMem(__hMem);
      }
      return 0;
}
```

7.10 Sha64AttachMemWithTag

The function allows attaching to an already allocated memory pool. When using this function it attaches to the memory with a given TAG allocated by Sha64AllocMemWithTag in any application or device driver. On success the returning value is ERROR_SUCCESS and the pointer „pUserVA“ for Windows function access and „pSystemVA“ for realtime task access becomes valid, as well as the physical memory address. Otherwise the returning value corresponds to that from GetLastError().

```
ULONG SHAAPI Sha64AttachMemWithTag(
          ULONG Tag,           //IN  : Memory TAG
          ULONG* pSize,        //OUT : Memory range
          PVOID* ppUserVA,     //OUT : Pointer to memory in user address space
          PVOID* ppSystemVA,   //OUT : Pointer to memory in system address space
          ULONG* pPA,          //OUT : Physical memory address
          HANDLE* phMem)       //OUT : Handle to memory device
```

Note:
When sharing memory between applications, each of the application first should check if any memory has been already allocated. Only if no memory was allocated before, the application should try to allocate memory with ShaAllocMemWithTag.

7.11 Sha64DetachMem

This function releases the attachement to a memory device. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG SHAAPI Sha64DetachMem(HANDLE hMem)  //IN : Handle to memory device
```

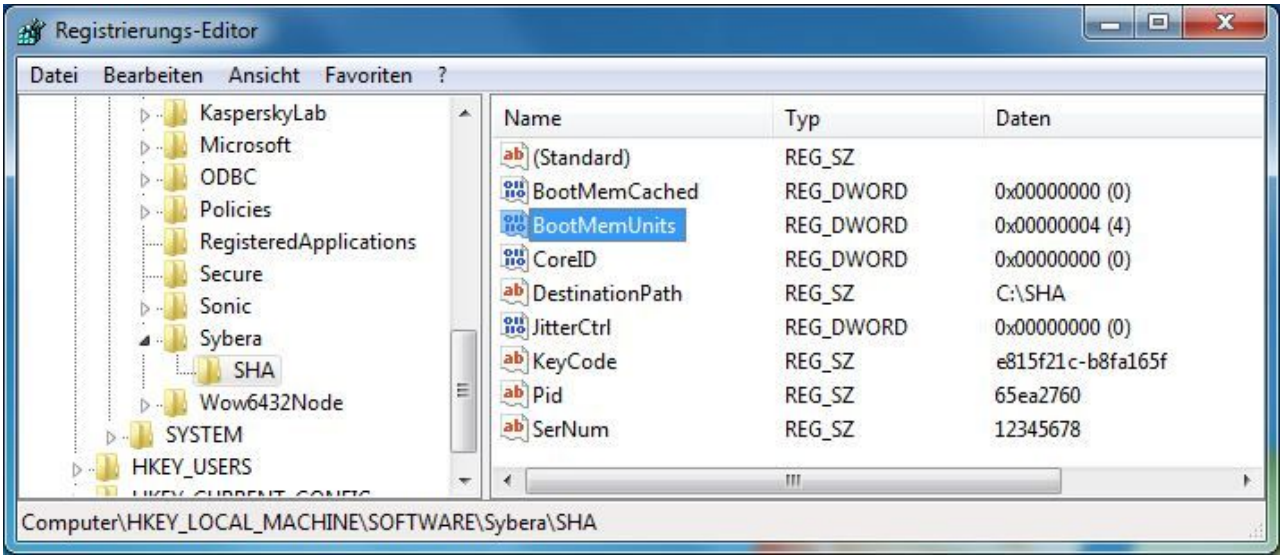
Sample:

```
//Declare some globals
PULONG      __pUserMem = NULL;
PULONG      __pSystemMem = NULL;
HANDLE      __hMem = NULL;


void TimerTask(void)
{
      //Update attached memory
      if (__hMem) { __pSystemMem[1]++; }
}


int main(int argc, char* argv[])
{
      ULONG Error = 0;
      HANDLE hProc = NULL;
      HANDLE hTask = NULL;
      ULONG MemBasePA;
      ULONG MemSize;

      //Try to attach to the first memory
      Error = Sha64AttachMemWithTag(
                        0x1234,                 //Memory tag
                        (ULONG*)&MemSize,       //Memory size
                        (void**)&__pUserMem,    //Pointer for Windows function
                        (void**)&__pSystemMem,  //Pointer for realtime task
                        &MemBasePA,             //Physical memory address
                        &__hMem);               //Handle to memory device

      if (Error == ERROR_SUCCESS)
      {
            //Init physical memory
            memset(__pUserMem, 0, MemSize);

            Error = Sha64CreateProcessor(REALTIME_PERIOD, FALSE, NULL, &hProc);
            Error = Sha64CreateTask(TimerTask, 1, 1000, FALSE, NULL, hProc,
                                                                  &hTask);
            Error = Sha64ControlProcessor(hProc, PROC_CTRL_CONTINUE, 0);

            //Wait for key press
            while (!_kbhit())
            {
                  printf("Press any key\r");
                  Sleep(500);
            }
            //Cleanup
            Sha64DestroyTask(hTask);
            Sha64DestroyProcessor(hProc);
            Sha64DetachMem(__hMem);
      }
      return 0;
}
```

7.12 BOOT Memory

SHA64 allows to allocate a large amount of physical contiguous memory at boot time, which may be shared between various applications and realtime tasks. When using boot memory, no application needs to play the role as a memory master, when sharing the memory with different processes. Boot memory is allocated in units of MB, upto 32 MB, depending on the system RAM. Attaching to boot memory requires the tag BOOT_MEM_TAG.

BootMemCached : Flag for cached memory (default off)
BootMemUnits: Memory Units in MB



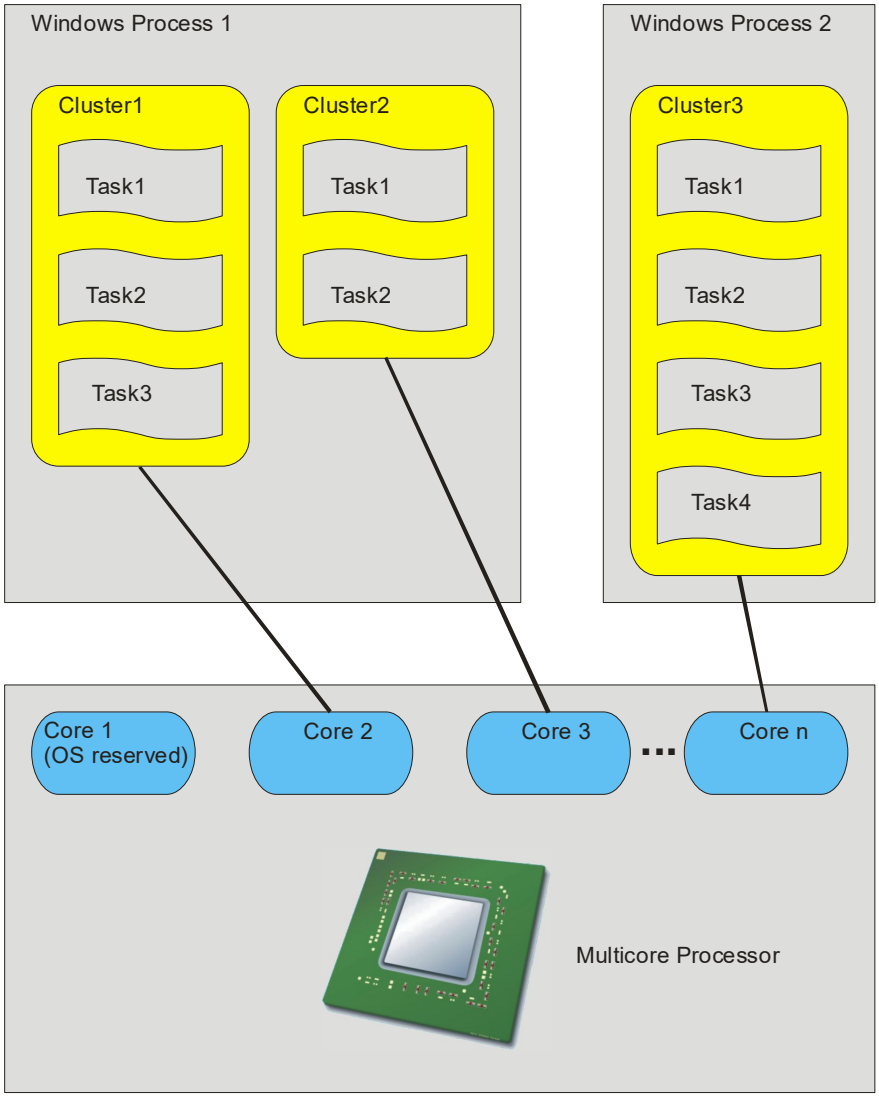
Note: Best timing results are achieved when activating memory cache. This is allways recommented, if the memory is not used by DMA devices

Sample:

```
PULONG      __pUserMem = NULL;
PULONG      __pSystemMem = NULL;
HANDLE      __hMem = NULL;


void TimerTask(void)
{
      //Update boot memory
      if (__hMem) { __pSystemMem[0] += 1; }
}


int main(int argc, char* argv[])
{
      ULONG Error = 0;
      HANDLE hProc = NULL;
      HANDLE hTask = NULL;
      ULONG MemBasePA;
      ULONG MemSize;

      //Try to attach to the first memory
      Error = Sha64AttachMemWithTag(
            BOOT_MEM_TAG,           //Memory tag
            (ULONG*)&MemSize,       //Memory size
            (void**)&__pUserMem,    //Pointer to user memory for Windows access
            (void**)&__pSystemMem,  //Pointer to user memory for Realtime access
            &MemBasePA,             //Physical memory address
            &__hMem);               //Handle to memory device

      if (Error == ERROR_SUCCESS)
      {
            //Init physical memory
            memset(__pUserMem, 0, MemSize);

            Error = Sha64CreateProcessor(REALTIME_PERIOD, FALSE, NULL, &hProc);
            Error = Sha64CreateTask(TimerTask, 1, 1000, FALSE, NULL, hProc, &hTask);
            Error = Sha64ControlProcessor(hProc, PROC_CTRL_CONTINUE, 0);

            //Wait for key press
            while (!_kbhit())
            {
                  printf("Press any key\r");
                  Sleep(500);
            }

            //Cleanup
            Sha64DestroyTask(hTask);
            Sha64DestroyProcessor(hProc);
            Sha64DetachMem(__hMem);
      }
      return 0;
}
```

8 Realtime Module

The SHA64 realtime engine allows handling of non-preemptive realtime multitask clusters on several processor cores. The realtime cluster runs independantly from the Windows Operating system with its own scheduling system. For data exchange it shares memory between the Windows process address space and the realtime cluster.



Note: The number of processors used by windows is scalable at installation

8.1 Power Management (recommended)

For proper operation, make shure within the BIOS the INTEL Speedstep Technologie, the INTEL TurboBoost Technologie as well as the INTEL C-STATE Technologie is turned off.

Enhanced SpeedStep - SpeedStep also modulates the CPU clock speed and voltage according to load, but it is invoked via another mechanism. The operating system must be aware of SpeedStep, as must the system BIOS, and then the OS can request frequency changes via ACPI. SpeedStep is more granular than C1E halt, because it offers multiple rungs up and down the ladder between the maximum and minimum CPU multiplier and voltage levels.

C1E enhanced halt state - Introduced in the Pentium 4 500J-series processors, the C1E halt state replaces the old C1 halt state used on the Pentium 4 and most other x86 CPUs. The C1 halt state is invoked when the operating system's idle process issues a HLT command. (Windows does this constantly when not under a full load.). C0 is the operating state. C1 (often known as Halt) is a state where the processor is not executing instructions, but can return to an executing state essentially instantaneously. All ACPI-conformant processors must support this power state. Some processors, such as the Pentium 4, also support an Enhanced C1 state (C1E or Enhanced Halt State) for lower power consumption. C2 (often known as Stop-Clock) is a state where the processor maintains all software-visible state, but may take longer to wake up. This processor state is optional. C3 (often known as Sleep) is a state where the processor does not need to keep its cache coherent, but maintains other state. Some processors have variations on the C3 state (Deep Sleep, Deeper Sleep, etc.) that differ in how long it takes to wake the processor. This processor state is optional.

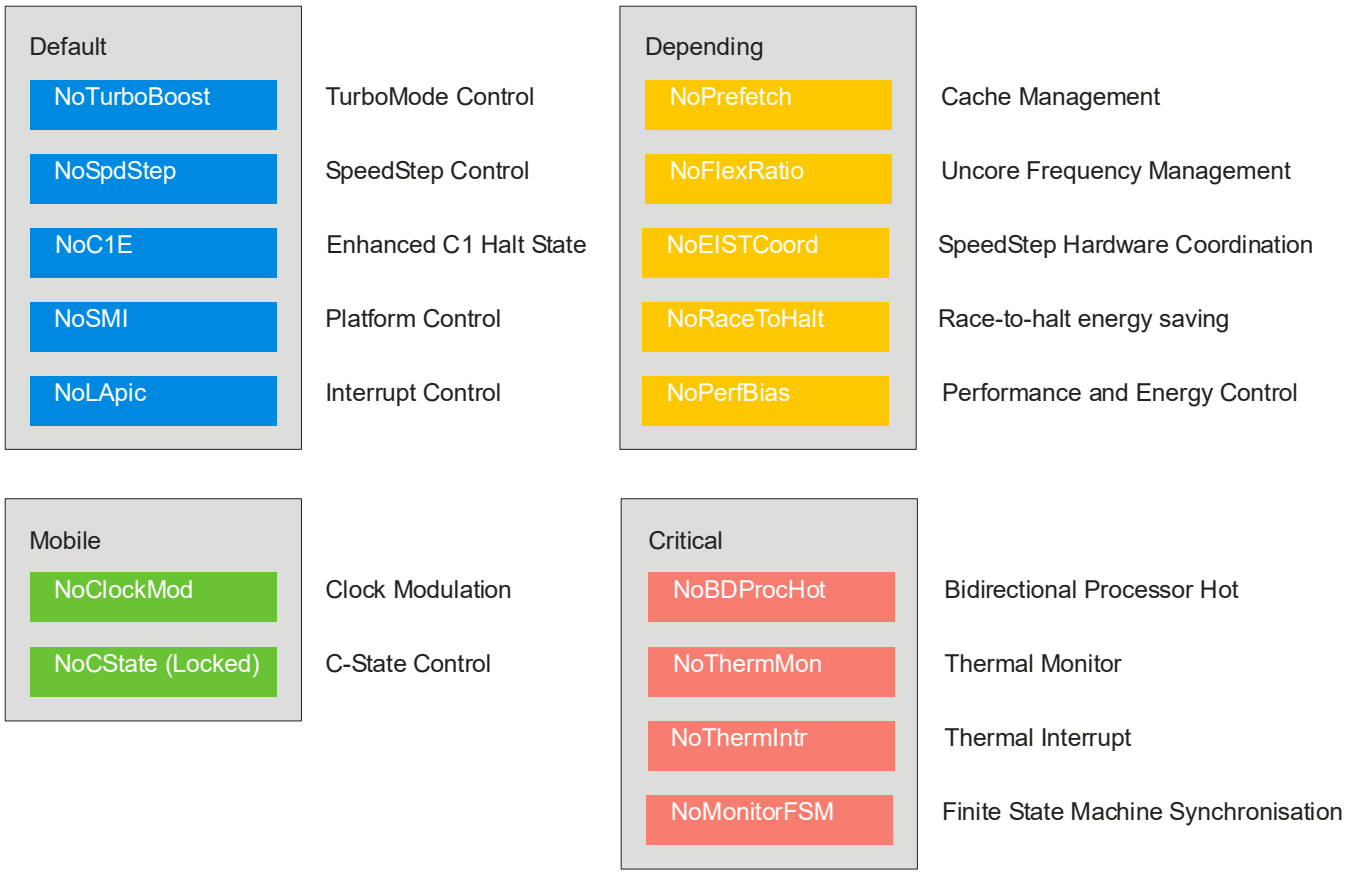
Intel® Turbo Boost Technology automatically allows processor cores to run faster than the base operating frequency, increasing performance. Under some configurations and workloads, Intel® Turbo Boost technology enables higher performance through the availability of increased core frequency. Intel® Turbo Boost technology automaticallyallows processor cores to run faster than the base operating frequency if the processor is operating below rated power, temperature, and current specification limits. Intel® Turbo Boost technology can be engaged with any number of cores or logical processors enabled and active. This results in increased performance of both multi-threaded and single-threaded workloads.

8.2 Active Jitter Control

Jitter control for realtime is the most complex issue to implement, since it differs from each platform and CPU. The realtime system allows the adaptive adjustment of various power management mechanism. Therefore 4 groups are available:

- Default: Default settings by installation
- Mobile: Additional settings for mobile platforms and CPUs
- Depending: Platform and CPU depending settings
- Critical: CPU life critical settings

Registry Values:



8.3 Adaptive Jitter Control

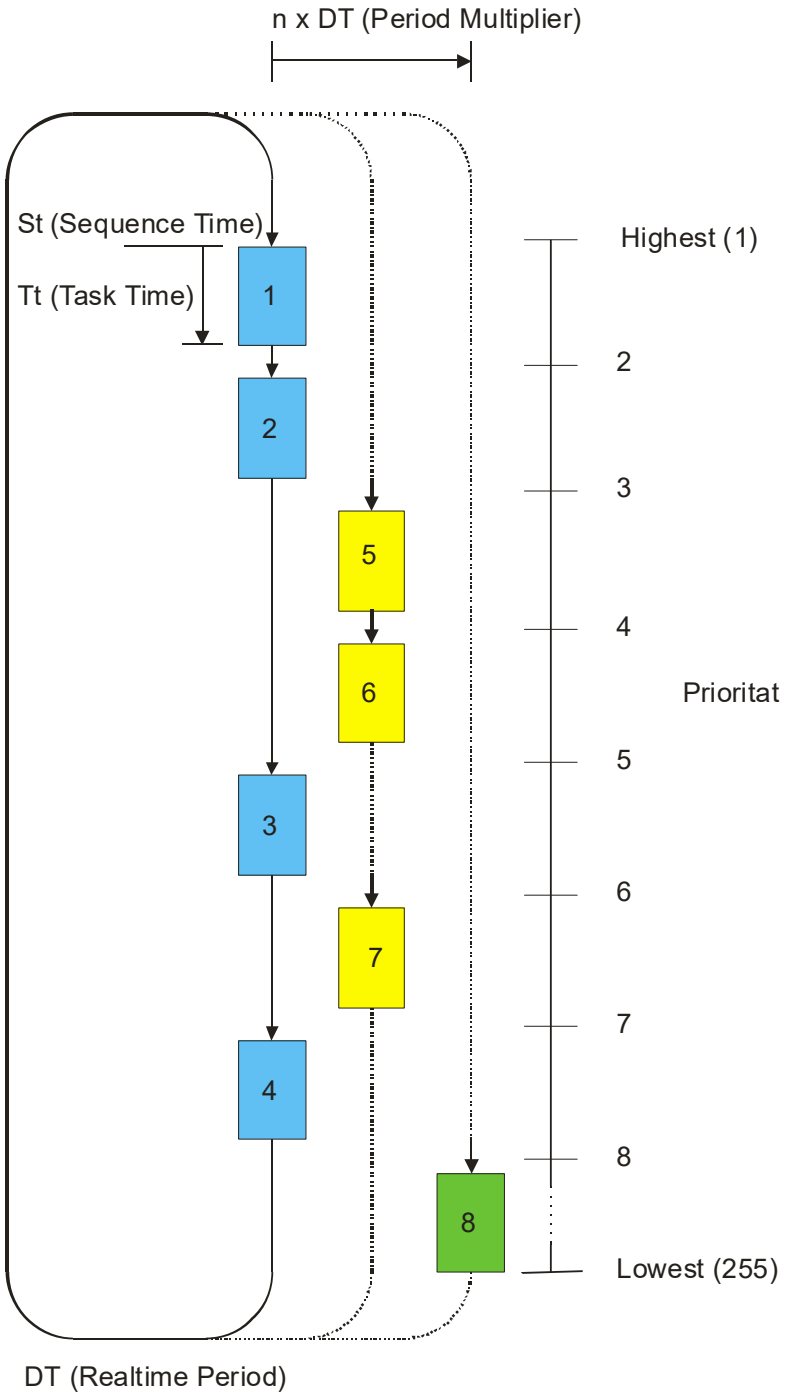
Adaptive Jitter Control realizes a typical jitter behaviour less than 5 µsec (depending on hardware platform). Adaptive Jitter Control is available for each realtime task, which measures the jitter divergation for each realtime loop and approaches a virtual period time by an adaptive delay. Thus an additional task load of 20µsec is required.

Sample:

```
//Create an exclusive processor device
//Create a task device for the processor (*** with jitter compensation ***)
//Start the processor scheduler
Error = Sha64CreateProcessor(REALTIME_PERIOD, FALSE, NULL, &__hProc);
Error = Sha64CreateTask(TimerTask, 1, 1, TRUE, NULL, __hProc, &hTask);
Error = Sha64ControlProcessor(__hProc, PROC_CTRL_CONTINUE, 0);
```

8.4 Realtime Task Cluster

With the SHA64 realtime system several tasks can be assigned to a processor core within a process address space. Each task itself can be scheduled by a period multiplier and a sequencial priority level.

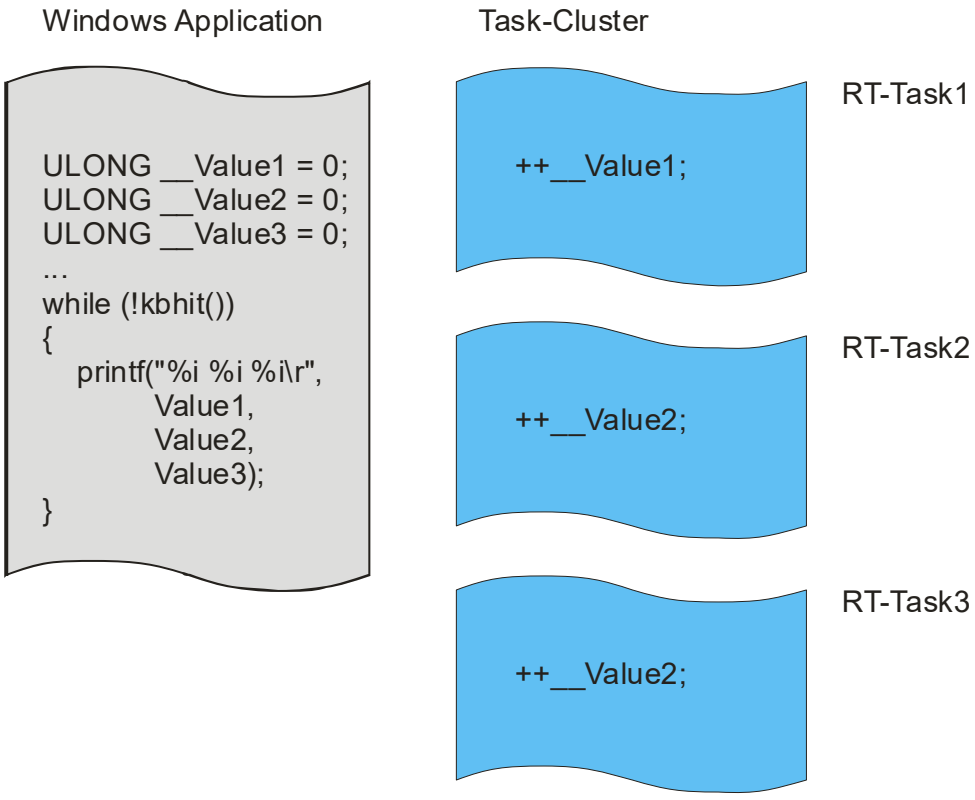


Sample

Task1:	Period = 1 * DT	St = 0
Task2:	Period = 1 * DT	St = Tt1
Task3:	Period = 1 * DT	(n=1) -> St = Tt1 + Tt2+ Tt5 + Tt6 (n>1) -> St = Tt1 + Tt2
Task4:	Period = 1 * DT	(n=1) -> St = Tt1 + Tt2 + Tt3 + Tt5 + Tt6 + Tt7 (n>1) -> St = Tt1 + Tt2 + Tt3
Task5:	Period = a * DT	(n=1) -> St = Tt1 + Tt2 (n>1) -> St = 0
Task6:	Period = a * DT	(n=1) -> St = Tt1 + Tt2 + Tt5 (n>1) -> St = Tt5
Task7:	Period = a * DT	(n=1) -> St = Tt1 + Tt2 + Tt3 + Tt5 (n>1) -> St = Tt5 + Tt6
Task8:	Period = b * DT	(n=1) -> St = Tt1 + Tt2 + Tt3 + Tt4 + Tt5 + Tt6 + Tt7 (n>1) (n == a) -> St = Tt5 + Tt6 + Tt7 (n != a) -> St = 0

8.5 Data Exchange

For data exchange the realtime cluster shares memory between the Windows process address space and the realtime cluster address space. The memory must be static.



Note:
As described before, memory allocated by Sha64AllocMemWithTag or attached by Sha64AttachMemWithTag may also be used for exchanging data between Windows function and realtime task.

8.6 Sha64CreateProcessor

If a free core is available, this routine enables a processor to act as a realtime cluster. Therfor the processor is booted up and set to 32 bit protected mode. The processor scheduler may act as a singleshot timer, or as a period cluster system. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64CreateProcessor(
            ULONG      Period,           //IN:  realtime period
            ULONG      ExceptMask,       //IN:  Exception Bit Mask (optional)
            BOOLEAN    bSingleShot,      //IN:  single shot flag
            FP_ADJUST* pfpAdjustFunc,    //OUT: Adjust function pointer (optional)
            HANDLE*    phProc)           //OUT: processor handle
```

Note:
The execution of the programming code of all active tasks may not exceed the realtime period. It must be considered that program jumps within the realtime routine may not block the realtime scheduler.

8.7 Sha64DestroyProcessor

This routine disables the realtime cluster and stops the processor core. This function has to be called before terminating the application. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64DestroyProcessor(HANDLE hProc) //OUT: processor handle
```

8.8 Sha64GetProcessorHandle

This routine allows another windows process to gain access to a processor cluster for control purposes. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64GetProcessorHandle(
                            ULONG Index,      //IN:  Processor index
                            HANDLE* phProc)   //OUT: Processor handle
```

8.9 Sha64GetProcessorInfo

This routine get information about a processor cluster. The processor cluster may running, or the processor resides in an exception state. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64GetProcessorInfo(
                  HANDLE hProc,           //IN:  Processor handle
                  PULONG pIndex,          //OUT: Processor index (optional)
                  PULONG pState,          //OUT: Processor state (optional)
                  PULONG pPeriod,         //OUT: Processor period (optional)
                  PULONG pLoopCnt,  //OUT: Cluster loop counter (optional)
                  PULONG pExceptNum,      //OUT: Exception number (optional)
                  PULONG pExceptCode,     //OUT: Exception code (optional)
                  PULONG pExceptRip,      //OUT: Exception RIP (optional)
                  PULONG pExceptRsp,      //OUT: Exception RSP (optional)
                  PULONG64 pApicFreq,     //OUT: APIC frequency (optional)
                  PULONG64 pCpuFreq)      //OUT: CPU frequency (optional)
```

8.10 Sha64ControlProcessor

After enabling the processor cluster it resides in WAIT state until a control command is sent. This allows first creation of all cluster tasks, followed by a synchronized start. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64ControlProcessor(
                        HANDLE  hProc,    //IN:  Processor handle
                        ULONG Control,    //IN:  Control command
                        ULONG DbgVA)      //IN:  Debug virtual address (optional)
```

8.11 Sha64CreateTask

This routine announces a task to a processor cluster. The task will be executed sequentially within the cluster, due to its priority priority (1 – 255) and period multiplier. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64CreateTask(
            FP_PROC_TASK fpProcTask,      //IN:  function pointer
            ULONG   Priority,             //IN:  priority
            ULONG   Multiplier,           //IN:  period multiplier
            HANDLE  hProc,                //IN:  processor handle
            BOOLEAN bJitterCtrl,          //IN:  adaptive jitter control
            PVOID   pContext,             //IN:  task context pointer
            HANDLE* phTask)               //OUT: task handle
```

Prototype:

```
typedef void(*FP_PROC_TASK)(PVOID);
```

Note:
A task is assigned to one processor core. All tasks assigned to a processor core must share the same windows process address space. However different processor cores may be assigned to different windows process address spaces.

8.12 Sha64DestroyTask

This routine removes the task from the realtime cluster. This function has to be called before terminating the realtime cluster. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64DestroyTask(HANDLE hTask)           //IN: task handle
```

8.13 Sha64GetTaskHandle

This routine allows another windows process to gain access to a cluster task for control purposes. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64GetTaskHandle(
                        HANDLE hProc)     //IN:  Processor handle
                        ULONG Index,      //IN:  Task index
                        HANDLE* phTask)   //OUT: Task handle
```

8.14 Sha64GetTaskInfo

This routine get information about a cluster task. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that from GetLastError().

```
ULONG Sha64GetTaskInfo(
            HANDLE hTask,                 //IN:  Task handle
            PULONG pIndex,                //OUT: Task index (optional)
            PULONG pMultiplier,           //OUT: Period multiplier (optional)
            FP_PROC_TASK fpProcTask*,     //OUT: Function pointer (optional)
```

Sample:

```
//Declare some globals
ULONG __TimerCnt[3] = { 0, 0, 0 };

void TimerTask1(void) { __TimerCnt[0]++; }
void TimerTask2(void) { __TimerCnt[1]++; }
void TimerTask3(void) { __TimerCnt[2]++; }

int main(int argc, char* argv[])
{
ULONG Error = 0;
HANDLE hProc[3] = { NULL, NULL, NULL };
HANDLE hTask[3] = { NULL, NULL, NULL };
ULONG Period = 0;
ULONG64 CpuFreq = 0;
ULONG PhyProcNum;

//Get number of physical processors
if (ERROR_SUCCESS != Sha64GetSystemInfo(&PhyProcNum, NULL))
    exit(0);

//Create an exclusive processor device
Error = Sha64CreateProcessor(100,FALSE,NULL,&hProc[0]);  //100 usec periodic
Error = Sha64CreateProcessor(1000,TRUE,NULL,&hProc[1]);  //1 ms single shot

//Create a cluster task (Processor: 1, Priority: 1, Multiplier: 1)
Error = Sha64CreateTask(TimerTask1, 1, 1, FALSE, NULL, hProc[0], &hTask[0]);

//Create a cluster task (Processor: 1, Priority: 2, Multiplier: 5)
Error = Sha64CreateTask(TimerTask2, 2, 5, FALSE, NULL, hProc[0], &hTask[1]);

//Create a cluster task (Processor: 2, Priority: 1, Multiplier: 10)
Error = Sha64CreateTask(TimerTask3, 1, 10, FALSE, NULL, hProc[1], &hTask[2]);

//Start the processor scheduler
Error = Sha64ControlProcessor(hProc[0], PROC_CTRL_CONTINUE, 0);
Error = Sha64ControlProcessor(hProc[1], PROC_CTRL_CONTINUE, 0);

//Get the CPU frequency
Sha64GetProcessorInfo(hProc[0],NULL,NULL,&Period,NULL,NULL,NULL,NULL,NULL,&CpuFreq)

//Wait for key press
while (!_kbhit()) { Sleep(100); }

//Cleanup
Sha64DestroyTask(hTask[0]);
Sha64DestroyTask(hTask[1]);
Sha64DestroyTask(hTask[2]);
Sha64DestroyProcessor(hProc[0]);
Sha64DestroyProcessor(hProc[1]);
return 0;
}
```

9 Event based Synchronisation

SHA64 provides an event based realtime synchronisation mechanism, to control realtime cluster tasks and windows application routines.

9.1 Sha64CreateEvent

This function creates a synchronisation event for realtime and non-realtime usage and an event function is provided for use within a cluster task. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that of GetLastError().

```
ULONG Sha64CreateEvent(
                  ULONG EventID,                //Event ID
                  FP_EVENT* pfpEventFunc,       //System event function pointer
                  HANDLE* phUserEvent,          //Windows event handle
                  HANDLE* phSystemEvent)        //Cluster event handle
```

Prototype:

```
typedef void(*FP_EVENT)(HANDLE, BOOLEAN);
```

9.2 Sha64DestroyEvent

This function destroys a synchronisation event. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that of GetLastError().

```
ULONG Sha64DestroyEvent(
                  HANDLE hUserEvent,            //OUT : Windows event handle
                  HANDLE hSystemEvent)          //OUT : Cluster event handle
```

Sample:

```
//Declare some globals
ULONG __TimerCnt = 0;
HANDLE __hUserEvent = NULL;
HANDLE __hSystemEvent = NULL;
FP_EVENT __fpEventFunc = NULL;


void TimerTask(void)
{
	__fpEventFunc(__hSystemEvent, TRUE);
	__TimerCnt++;
}


int main(int argc, char* argv[])
{
	ULONG Error = 0;
	HANDLE hProc = NULL;
	HANDLE hTask = NULL;
	ULONG Period = 0;
	ULONG PhyProcNum;

	//Get number of physical processors
	if (ERROR_SUCCESS != Sha64GetSystemInfo(&PhyProcNum, NULL))
		exit(0);

	//Create system event
	Sha64CreateEvent(1, &__fpEventFunc, &__hUserEvent, &__hSystemEvent);

	//Create an exclusive processor device (5sec, single shot)
	//Create a task device for the processor (Priority: 1, Multiplier: 1)
	//Start the processor scheduler
	Error = Sha64CreateProcessor(5000 * 1000, TRUE, NULL, &hProc);
	Error = Sha64CreateTask(TimerTask, 1, 1, FALSE, NULL, hProc, &hTask);
	Error = Sha64ControlProcessor(hProc, PROC_CTRL_CONTINUE, 0);

	//Wait for key press
	while (!_kbhit())
	{
		if (WAIT_OBJECT_0 == WaitForSingleObject(__hUserEvent, 0))
			printf("Event Test OK\n");

		//Do some delay
		Sleep(100);
	}

	//Cleanup
	Sha64DestroyTask(hTask);
	Sha64DestroyProcessor(hProc);
	Sha64DestroyEvent(__hUserEvent,	__hSystemEvent);
	return 0;
}
```

10 Timing Control

10.1 Dynamic Frequency Control

Dynamic Frequency Control allows to adjust the realtime period within the realtime task. Therefore a function pointer can be obtained from the function Sha64CreateProcessor.

Prototype:

```
void(*FP_ADJUST)(HANDLE, ULONG);
```

Sample:

```
//Declare some globals
HANDLE      __hProc = NULL;
FP_ADJUST   __fpAdjustFunc = NULL;
ULONG       __Period = REALTIME_PERIOD;
ULONG64     __CpuFreq = 0;
ULONG64     __TimerCnt = 0;
__int64     __DiffCnt = 0;
__int64     __Tsc = 0;


void RealtimeTask(void)
{
      //Read CPU time stamp count and get period count
      __int64 tsc  = __rdtsc();
      __int64 PeriodTsc = (REALTIME_PERIOD * (__int64)__CpuFreq) / (1000 * 1000);

      if ((__Tsc) && (__CpuFreq))
      {
            __DiffCnt = tsc - __Tsc;

            //Compensate period drift
            if (__DiffCnt > PeriodTsc )   { __Period--; }
            else                          { __Period++; }

            //Adaptive drift compensation
            __fpAdjustFunc(__hProc, __Period);
      }

      //Save CPU time stamp count
      __Tsc = tsc;
}
```

10.2 Sha64SystemDelay

This function is exported from the library SHA64TIME.OBJ. It provides a micro second delay for use within a cluster task. As parameter the CPU frequency is required (available from Sha64GetProcessorInfo).

```
ULONG Sha64SystemDelay(
                  ULONG64 usec,     //IN:  Delay time [µsec]
                  ULONG64 CpuFreq)  //IN:  CPU frequency
```

Note:
The delay time must not exceed the realtime period

Sample:

```
void TimerTask(void)
{
	__t1 = __rdtsc();
	Sha64SystemDelay(10, __CpuFreq);
	__t2 = __rdtsc() - __t1;
}


int main(int argc, char* argv[])
{
	ULONG Error = 0;
	HANDLE hProc = NULL;
	HANDLE hTask = NULL;
	ULONG Period = 0;
	ULONG PhyProcNum;

	//Get number of physical processors
	if (ERROR_SUCCESS != Sha64GetSystemInfo(&PhyProcNum, NULL))
		exit(0);

	//Create an exclusive processor device
	//Create a task device for the processor
	//Start the processor scheduler
	Error = Sha64CreateProcessor(5000 * 1000, FALSE, NULL, &hProc);
	Error = Sha64CreateTask(TimerTask, 1, 1, FALSE, NULL, hProc, &hTask);
	Error = Sha64ControlProcessor(hProc, PROC_CTRL_CONTINUE, 0);

	//Get the CPU frequency
	Sha64GetProcessorInfo(hProc, NULL, NULL, &Period, NULL, NULL, NULL, NULL,
				NULL, NULL, &__CpuFreq);

	//Wait for key press
	while (!_kbhit())
	{
		printf("Press any key ... [%f]\r", (float)((__t2 - __t1) / __CpuFreq));

		//Do some delay
		Sleep(100);
	}

	//Cleanup
	Sha64DestroyTask(hTask);
	Sha64DestroyProcessor(hProc);
	return 0;
}
```

11 Silent Mode

The problem of the jitter behavior in conjunction with Windows and real-time is mainly due to the necessary bus synchronization between the processors. For example, jitter deviations of more than 100 µsec are possible on some PCs, despite the most complex platform control. With the real-time silent mode, Windows is selectivly put into sleep and woken up by program control. This allows a stable real-time behavior below 5 microseconds.

11.1.1 Sha64AcquireSilent

This function acquires the silent mode. Therefore a function pointer is returned for use inside the realtime task. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that of GetLastError().

Prototype:

```
ULONG Sha64AcquireSilent(OUT FP_SILENT* pfpSilentFunc);
```

11.1.2 Sha64ReleaseSilent

This function releases the silent mode. On success the returning value is ERROR_SUCCESS, otherwise the returning value corresponds to that of GetLastError().

Prototype:

```
ULONG Sha64ReleaseSilent(void);
```

11.1.3 FP_SILENT fpSilentFunc

This function is used to control the silent mode within the realtime task. If the flag bInit is set, the initial count value is taken over. The initial count value 0 stops the silent mode. The function returns the current count, decreased on every call. If the count reaches 0, the silent mode stops.

Prototype:

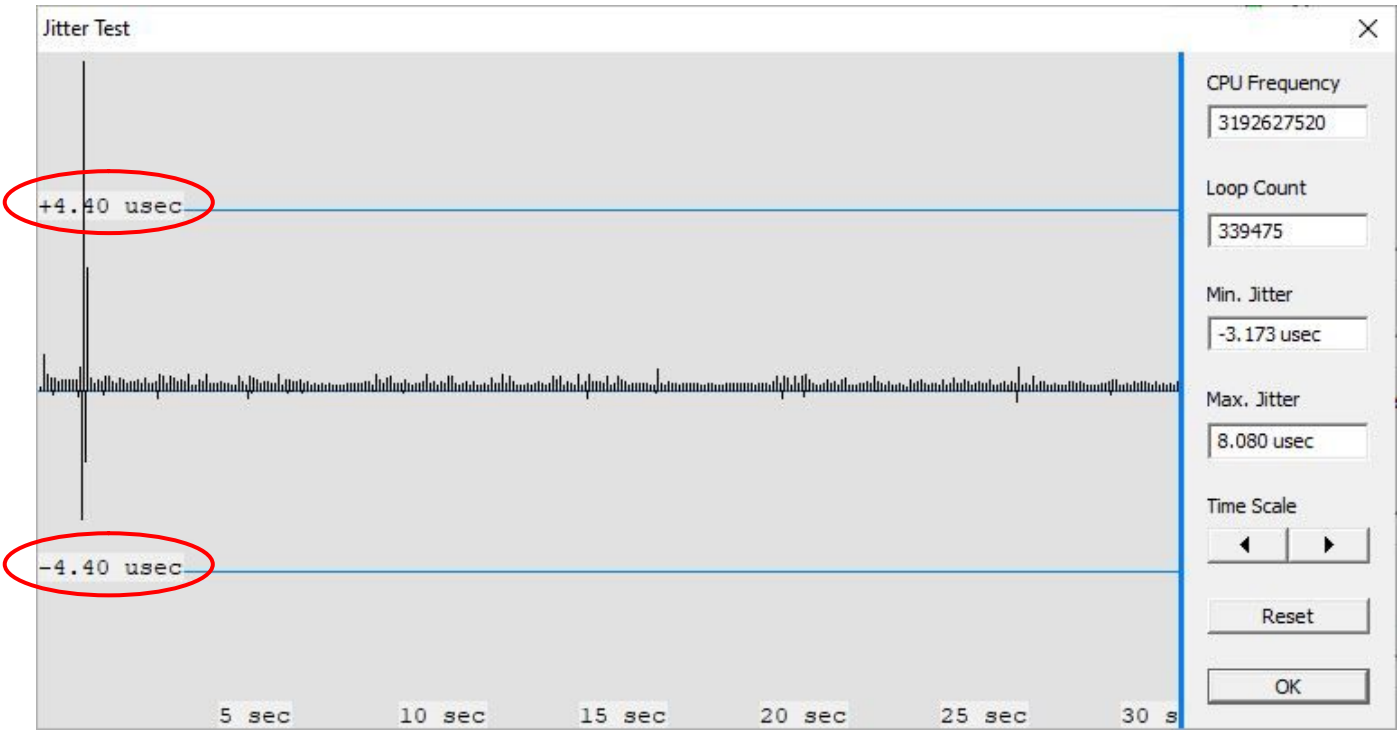
```
ULONG64 CurrentCount = fpAdjustSilent(
                            ULONG64 InitCount,      //Initial Count
                            BOOLEAN bInit);         //Initial Flag
```

Sample:

```
static ULONG64 __LoopCnt = 0;
FP_SILENT __fpSilentFunc = NULL;

void RtTask(PVOID)
{
	//Start silent mode after 5 sec (Realtime Period 100 µsec)
	if (__LoopCnt == 50000)
	{
		//Init silent mode for 10 sec (Realtime Period 100 µsec)
		__fpSilentFunc(100000, TRUE);
	}

	//Check silent counter. It Stops, if the return value reaches 0
	if (__fpSilentFunc(0, FALSE))
	{
		//Do silent code
		…
	}

	if (__LoopCnt == 400000)
	{
		//Stop silent mode immediately (if not stopped yet)
		__fpSilentFunc(0, TRUE);
	}

	//Increase loop counter
	__LoopCnt++;
}


int main(int argc, char* argv[])
{
	HANDLE hProc = NULL;
	HANDLE hTask = NULL;

	//Prepare silent mode and get silent mode function pointer
	Error = Sha64AcquireSilent(&__fpSilentFunc);

	Error = Sha64CreateProcessor(REALTIME_PERIOD, 0, FALSE, NULL, &hProc);
	Error = Sha64CreateTask(RtTask, 1, 1, FALSE, NULL, hProc, &hTask);
	Error = Sha64ControlProcessor(hProc, PROC_CTRL_CONTINUE, 0);

	…

	Sha64ControlProcessor(hProc, PROC_CTRL_HALT, 0);
	Sha64DestroyTask(hTask);
	Sha64DestroyProcessor(hProc);

	//Release silent mode
	Sha64ReleaseSilent();
	return 0;
}
```

11.2 USB Silent Mode

In cases, the USB enhanced facilities aren't offered to be turned off by the BIOS, SYBERA brought out a technique called "USB Silent Mode". Similar to the already known Silent Mode, where the whole operating system is turned off for a certain period of time, the "USB Silent Mode" turns off the Host Controller
Until any mouse or keyboard event occurs. The benefit thereby is the active display. The improvement of the jitter behavior is obviously:



Sample (UsbSilent.cpp):

```
#include "stdafx.h"
#include <Windows.h>
#include <conio.h>
#include <stdio.h>
#include <intrin.h>
#include "c:\sha\sha64exp.h"
#include "c:\sha\globdef64.h"
#include "c:\sha\sha64usb.h"

#pragma comment( lib, "c:\\sha\\sha64dll.lib" )

#define REALTIME_PERIOD      100

//***************************
USBMAP_XHCI_INIT_ELEMENTS();  //Init USB mapping elements
//***************************

//Declare some globals
ULONG     __LoopCnt = 0;
HANDLE    __hProc = NULL;


void RtTask(PVOID)
{
//****************************************
// Sart USB Silent Mode
//****************************************

      //Do some delay
      if (__LoopCnt == 100)
      {
            //Start silent mode
            USB_SYSTEM_XHCI_ENABLE_ALL_INTERRUPTS(FALSE);
      }

//****************************************
// Stop USB Silent Mode
//****************************************
      //Check for any USB events
      BOOLEAN bEvent = FALSE;
      USB_SYSTEM_XHCI_CHECK_ANY_EVENT(&bEvent);
      if (bEvent)
      {
            //Stop silent mode
            USB_SYSTEM_XHCI_ENABLE_ALL_INTERRUPTS(TRUE);

            __LoopCnt = 0;
      }

      //Increase loop counter
      __LoopCnt++;
}
```

```
int main(int argc, char* argv[])
{
      ULONG Error = 0;
      HANDLE hTask = NULL;

//******************
USBMAP_XHCI_ENTER(0x2100, 0);       //Map XHCI controllers (MapSize, MapOffs)
//******************

      //Prepare silent mode
      //Create an exclusive processor device
      //Create a task device for the processor (*** with jitter compensation ***)
      //Start the processor scheduler
      Error = Sha64CreateProcessor(REALTIME_PERIOD, 0, FALSE, NULL, &__hProc);
      Error = Sha64CreateTask(RtTask, 1, 1, FALSE, NULL, __hProc, &hTask);
      Error = Sha64ControlProcessor(__hProc, PROC_CTRL_CONTINUE, 0);

      //Wait for key press
      While (!kbhit())
      {
            printf("Press any key … Loop:[%5i]\r", __LoopCnt);

            //Do some delay
            Sleep(100);
      }

      //Cleanup
      Sha64ControlProcessor(__hProc, PROC_CTRL_HALT, 0);
      Sha64DestroyTask(hTask);
      Sha64DestroyProcessor(__hProc);

//******************
USBMAP_XHCI_EXIT();     //Unmap XHCI controllers
//******************

      return 0;
}
```

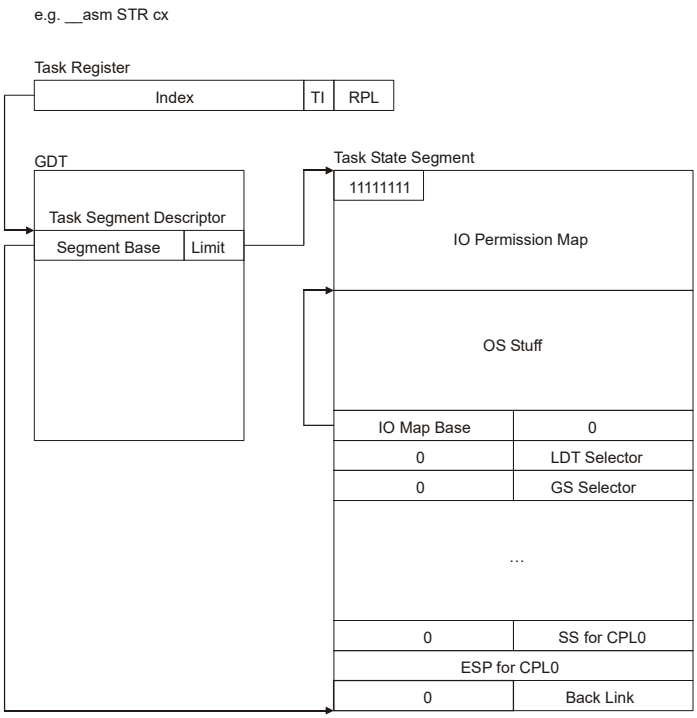
12 SHA64 and Windows PNP

One of the most surprising issue is how Windows controls IO-Port access. Considering older operating systems accessing IO-Ports was simply done at USER-Level. Under Windows accessing IO-Ports was done only at KERNEL-Level, and accessing IO-Ports under Windows depends on the Plug&Play configuration. All this behaviour is founded in the way how the protected mode of the x86 CPU works in conjunction with the operating system.

There are two ways the operating system is able to deny accessing IO Ports:

One mechanism is called IOPL Flag. To access IO ports the IOPL-Flag (IO privilege level) must match the condition of the CPL (current privilege level) of the code page descriptor. While USER-Mode code pages typically are running with CPL = 3, KERNEL-Mode pages are running with CPL = 0. Only if the condition CPL <= IOPL is TRUE, then access to IO port is possible.This was implemented in Windows 95 (IOPL = 3, IO port access is possible at USER-Mode level) and Windows (IOPL = 0, IO port access is possible at KERNEL-Mode level).

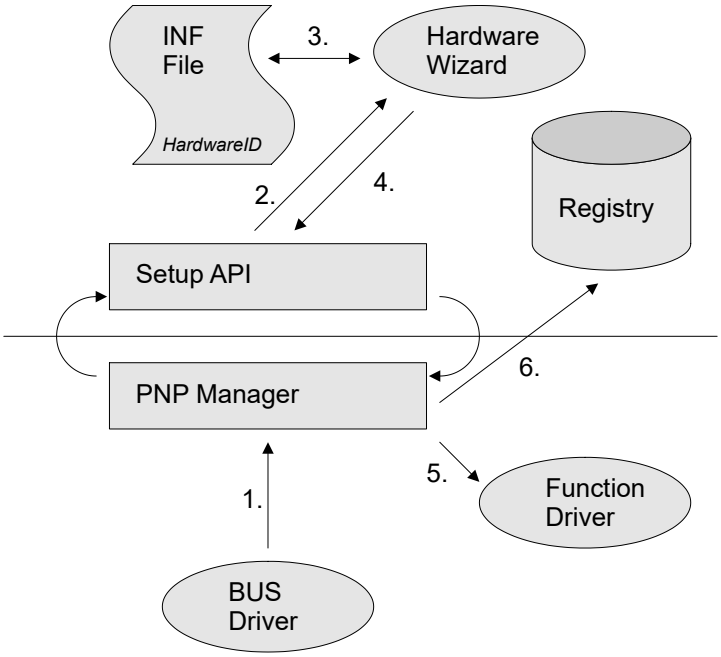
Now the surprising part came with Windows. Even when running at KERNEL-Mode (e.g. Device Drivers) access to IO ports with legacy drivers seems not to be possible any more. A lot of legacy drivers (e.g. NT drivers) do not work any more under Windows. Especially legacy drivers for PCI adapters with own resource management get into troubles. The reason therefor is a second mechanism implemented into the protected mode of the x86 CPU, called „IO Permission Map“ (IOMAP). The IOMAP is a part of the Task State Segment which is managed by the operating system.



12.1 PNP resource management

Of course to bypass this restriction of the OS we could change the IO Map Base of the corresponding Task State Segment. But there is an easier way when understanding how Windows maintenance the IOMAP.

In contrary to legacy devices, PNP devices require a dynamic resource management. This management relies on the corresponding INF-File of a PNP-Driver (WDM Driver) which provides a HardwareID (e.g. for PCI the HardwareID consists of VendorID and DeviceID) to identify the PNP device resources to use. Windows provides a BUS-Driver which scans the PNP devices and compares the found configuration with the given HardwareID of the INF-File. If both matches then Windows enables the IOMAP for the found device resources.



1. BUS-Driver scans the PNP devices and finds a new device
2. The SetupAPI calls the Hardware-Wizard
3. The Hardware Wizard reads in the corresponding INF-File
4. The HardwareID is given to the System
5. The AddDevice routine of the Function-Driver will be called
6. The device resources will be reported and the corresponding IOMAP enabled

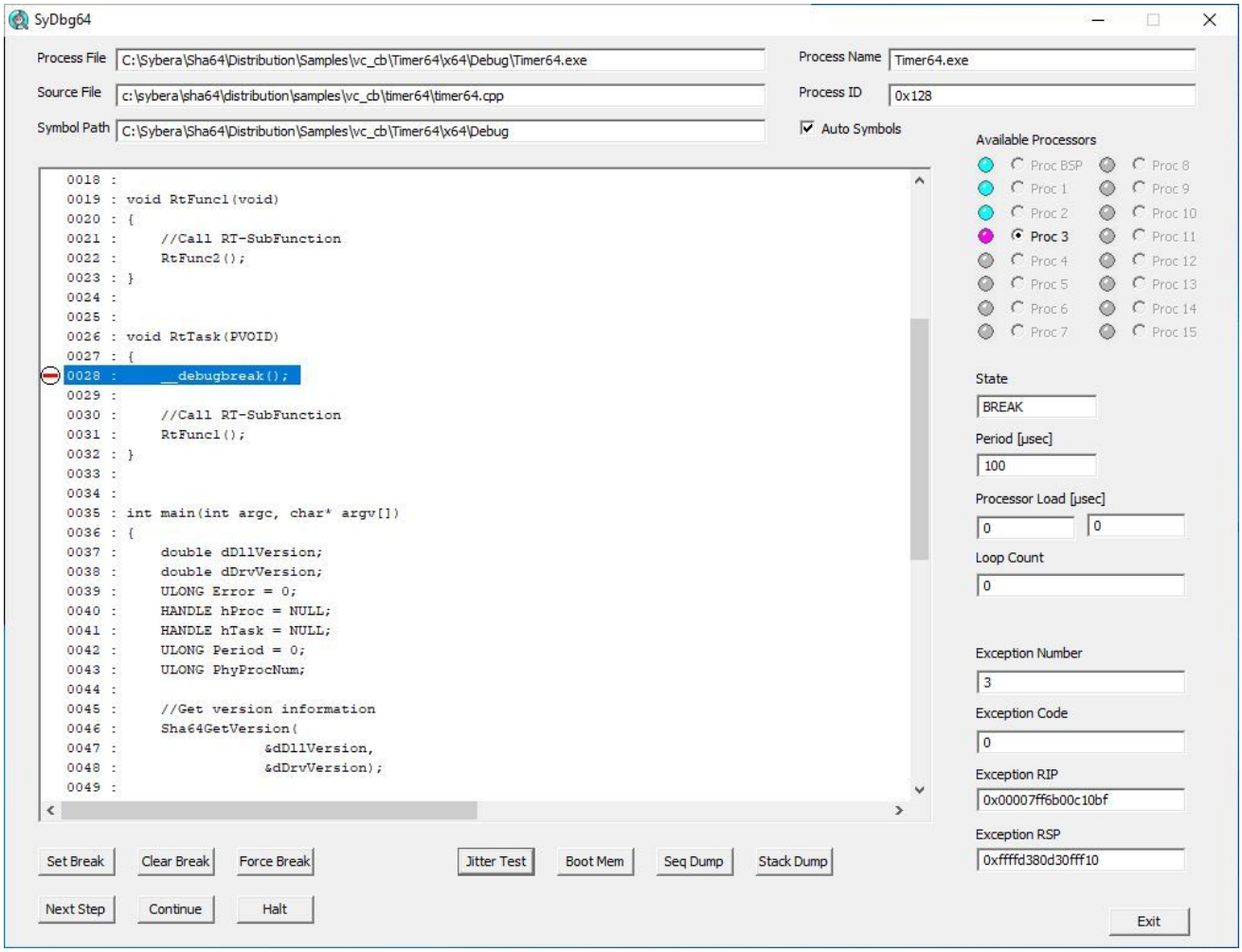
12.2 INF-Files

Since a PNP device driver is required for the hardware device, a so called support driver (SHA64SUPP.SYS) has to be installed with a corresponding INF-File containing the hardware ID. Here's a sample for such an INF-File (e.g. SHA64SUPP.INF):

```
[Version]
Signature="$WINDOWS NT$"
Class=%ClassName%
ClassGUID={DCA926B8-EE7A-41b5-BCB5-1BBAB7BD699C}
Provider=%Provider%
DriverVer=12/30/2011,1.0.1.0
CatalogFile=sha32supp.cat

[SourceDisksNames]
1=%Media%

[SourceDisksFiles]
Sha64Supp.sys = 1
Sha64Supp.inf = 1
insthlp.dll   = 1

[DestinationDirs]
ProductFiles = 12

;---------------- Product install ------------------

[Manufacturer]
%Mfg% = Sybera,NTx86

;*******************************************************
;This section needs to be changed adding the corresponding
;hardware ID
;*******************************************************
[Sybera.NTamd32]
%Product%=ProductInstall,PCI\VEN_10B5&DEV_3001
%Product%=ProductInstall,PCI\VEN_10B5&DEV_9050
%Product%=ProductInstall,PCI\VEN_001C&DEV_0001
%Product%=ProductInstall,PCI\VEN_10EC&DEV_8169

[ProductInstall]
CopyFiles=ProductFiles

[ProductFiles]
Sha64Supp.sys

[ProductInstall.Services]
AddService = Sha64Supp,2,ProductService
```

13 Debug and Control

SHA64 provides a debug and control application called SYDBG32. This application allows controlling the processor cores, as well as debugging.



Processor conditions:

Blue:	processor reserved for windows
Green:	processor used for realtime cluster
Violet:	processor in break state
Red:	processor in halt state
Grey:	processor not available

Note: To break into code, initial breakpoints must be set by the macro __debugbreak (hard coded breakpoint), set inside VisualStudio. Further breakpoints then may be set in SYDBG.

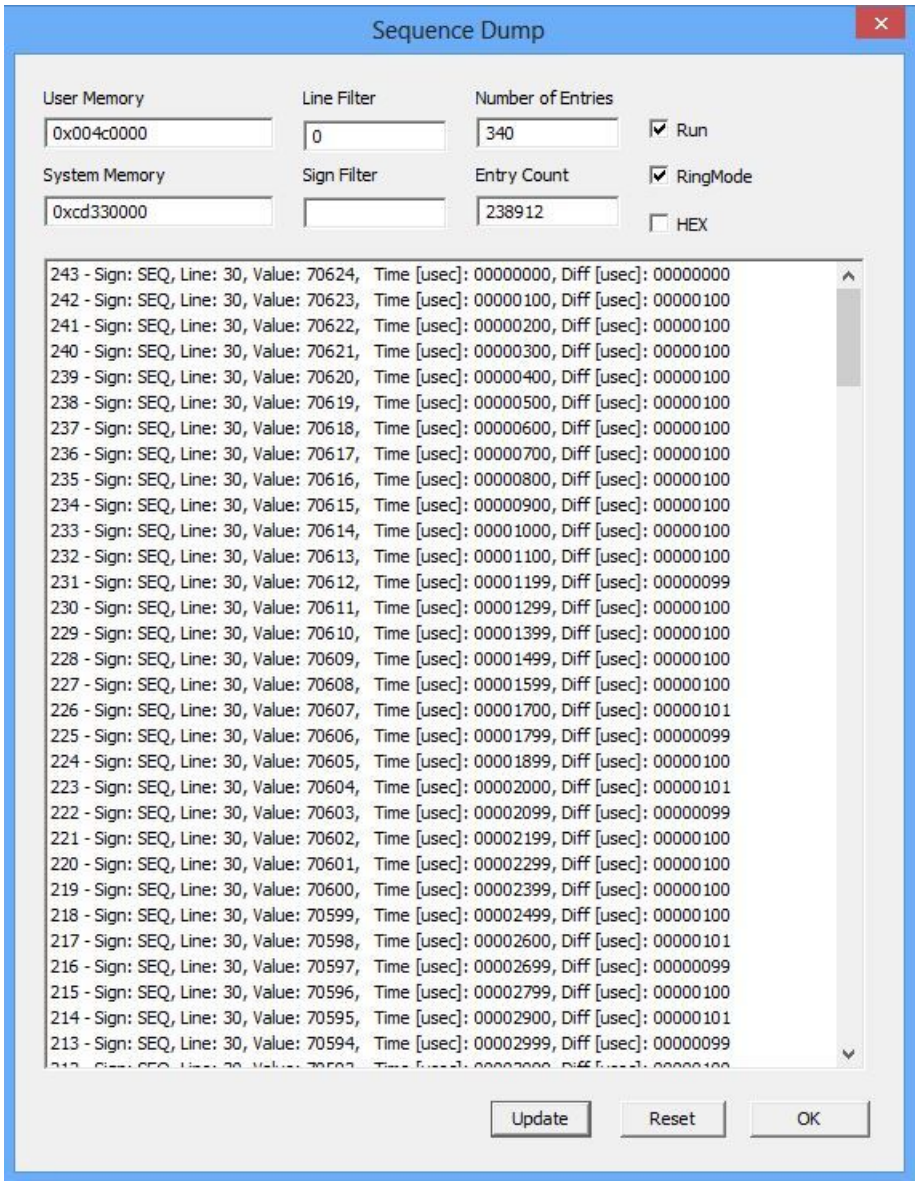
13.1 Sequence Analysis

SYDBG provides a possibility to keep track on hardcoded checkpoints whithout breaking into the code. Thus the timing behaviour remains untouched. To enable the sequence analysis, the source code need to be extended:

Sample:

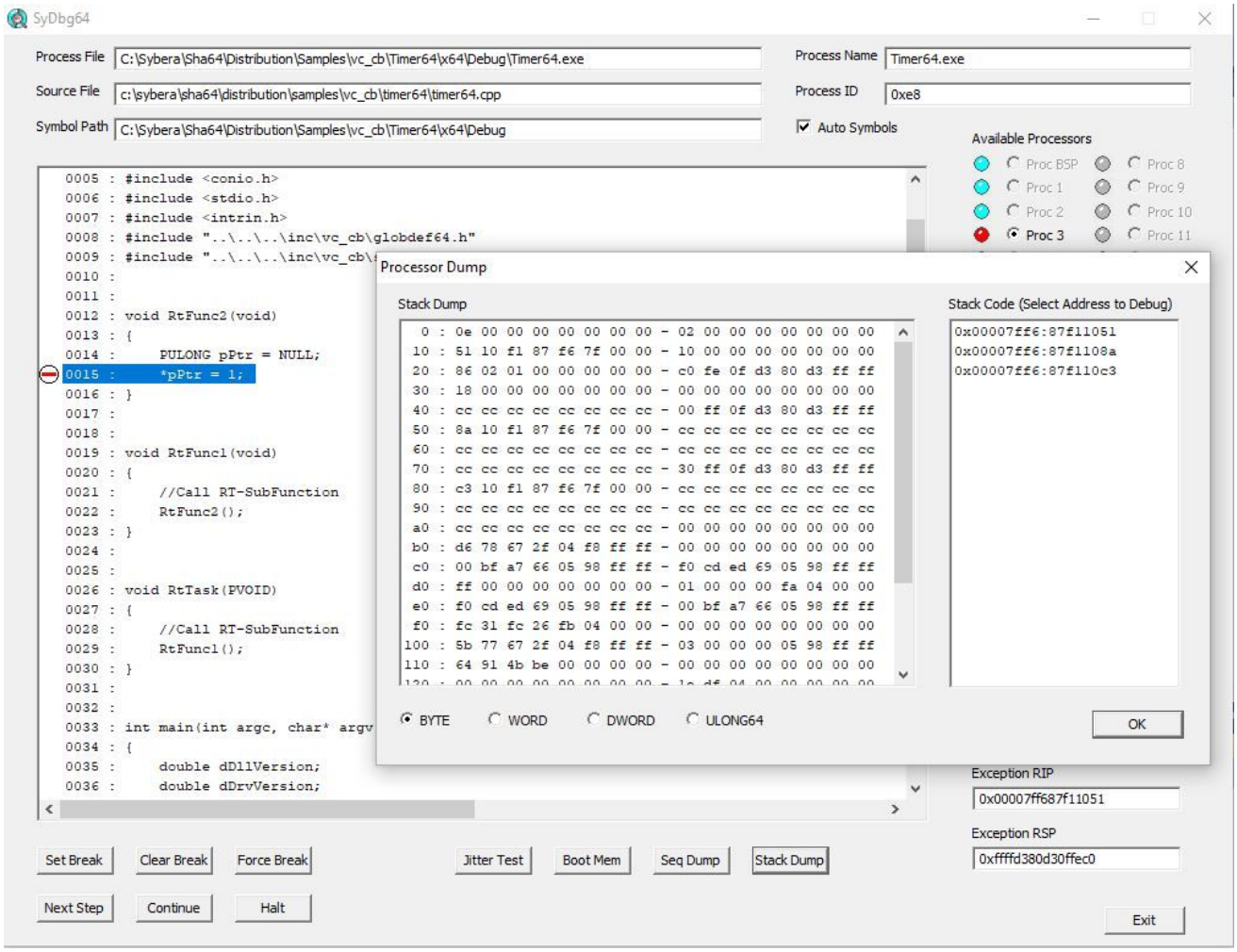
```
#define REALTIME_PERIOD      100

//**************************
SEQ_INIT_ELEMENTS(); //Init sequencing elements
//**************************

//Declare some globals
ULONG __TimerCnt = 0;


void TimerTask(PVOID)
{
      __TimerCnt++;

      //****************************
      SYSTEM_SEQ_CAPTURE("SEQ",__LINE__,__TimerCnt,TRUE);  //Set sequence capture
      //****************************
}


int main(int argc, char* argv[])
{
      double dDllVersion;
      double dDrvVersion;
      ULONG Error = 0;
      HANDLE hProc = NULL;
      HANDLE hTask = NULL;
      ULONG PhyProcNum;

//******************
SEQ_ATTACH();                        //Attach to sequence memory
SEQ_RESET(TRUE, TRUE, NULL, 0);           //Reset/Init sequence memory
//******************

      //Create an exclusive processor device
      //Create a task device for the processor
      //Start the processor scheduler
      Error = Sha32CreateProcessor(REALTIME_PERIOD, 0, FALSE, NULL, &hProc);
      Error = Sha32CreateTask(TimerTask, 1, 1, FALSE, NULL, hProc, &hTask);
      Error = Sha32ControlProcessor(hProc, PROC_CTRL_CONTINUE, 0);

      //Wait for key press
      while (!_kbhit())
      {
            printf("Press any key ... [%I64i]\r", __TimerCnt);
```

```
            //Do some delay
            Sleep(100);
      }
      //Cleanup
      Sha32DestroyTask(hTask);
      Sha32DestroyProcessor(hProc);

//******************
SEQ_DETACH(); //Detach from sequence memory
//******************

      return 0;
}
```



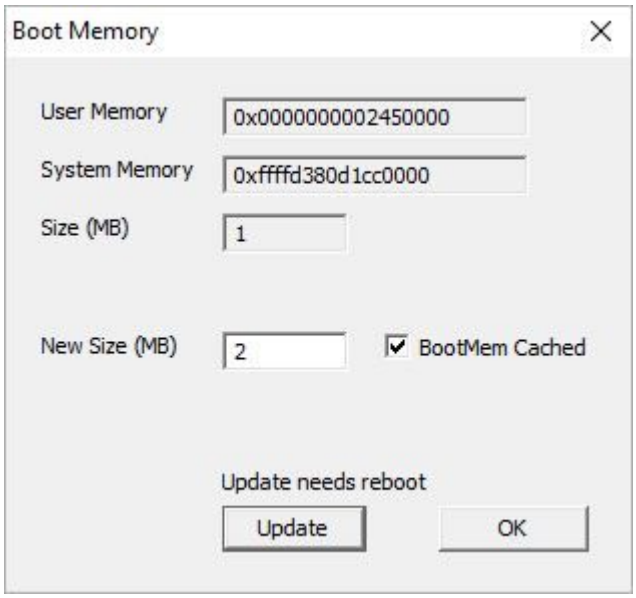
13.2 Stack Analysis

The integrated stack analysis makes it possible to track the jump addresses, and thus the history of the error. If the displayed stack code address is clicked, the associated source code will also be displayed. This means that even difficult errors in real-time code can now be found quickly.



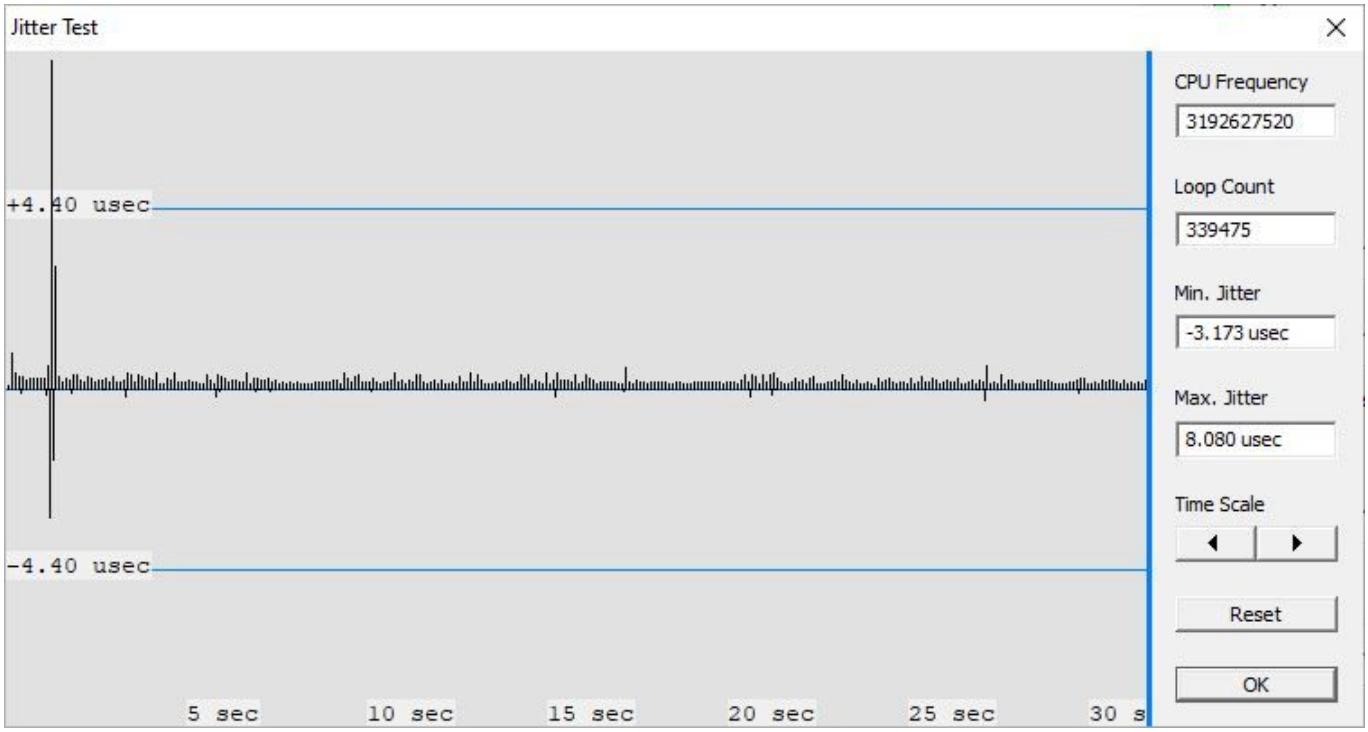
13.3 Boot Memory Manager

The boot memory manager allows adjusting the BOOT memory without entering the registry



13.4 Jitter Test

The new Jitter test facility allows measurement of the jitter behaviour at real-time over a period of time. This allows a high accuracy evaluation of the underlaying hardware.



14 Appendix

14.1 Function Overview

Exported Funktions	*Library*
Basic Module	
Sha64GetSystemInfo	SHA64DLL.DLL
Sha64GetVersion	SHA64DLL.DLL
Sha64LogError	SHA64DLL.DLL
Memory Module	
Sha64ReadPort	SHA64DLL.DLL
Sha64WritePort	SHA64DLL.DLL
Sha64SystemReadPortUchar	SHA64TIME.OBJ
Sha64SystemReadPortUshort	SHA64TIME.OBJ
Sha64SystemReadPortUlong	SHA64TIME.OBJ
Sha64SystemWritePortUchar	SHA64TIME.OBJ
Sha64SystemWritePortUshort	SHA64TIME.OBJ
Sha64SystemWritePortUlong	SHA64TIME.OBJ
Sha64MapMem	SHA64DLL.DLL
Sha64UnmapMem	SHA64DLL.DLL
Sha64AllocMemWithTag	SHA64DLL.DLL
Sha64FreeMem	SHA64DLL.DLL
Sha64AttachMemWithTag	SHA64DLL.DLL
Sha64DetachMem	SHA64DLL.DLL
Sha64GetDevInfo	SHA64DLL.DLL
Realtime Module	
Sha64CreateProcessor	SHA64DLL.DLL
Sha64DestroyProcessor	SHA64DLL.DLL
Sha64GetProcessorHandle	SHA64DLL.DLL
Sha64GetProcessorInfo	SHA64DLL.DLL
Sha64ControlProcessor	SHA64DLL.DLL
Sha64CreateTask	SHA64DLL.DLL
Sha64DestroyTask	SHA64DLL.DLL
Sha64GetTaskHandle	SHA64DLL.DLL
Sha64GetTaskInfo	SHA64DLL.DLL
Sha64CreateEvent	SHA64DLL.DLL
Sha64DestroyEvent	SHA64DLL.DLL
Sha64AcquireSilent	SHA(64)DLL.DLL
Sha64ReleaseSilent	SHA(64)DLL.DLL
Sha64SystemDelay	SHA64TIME.OBJ