Kithara | The Pulse of Real-Time

White Paper

Future-Proof Software Architecture with Kithara RealTime Suite

Going forward with Dedicated Mode and KiK64

Kithara RealTime Suite is a real-time extension for Windows operating systems. It is a modular software that, from an outside perspective, has only gone through a small amount of changes during its more than 20 years of existence. Whenever internal adjustments had to be made due to external developments of the underlying operating systems or hardware platforms, major changes for the user were generally avoided. This is an important factor, as the compatibility with earlier versions of the software has always been a high priority.

As a consequence, several alternative approaches to different issues were supported and therefore the variety of solutions increased continuously. Among them, Kithara RealTime Suite over the years, had to make distinctions regarding hardware, operating system, bit size of the different program parts, interrupt handling context as well as mode of real-time execution. Due to the increasing number of alternatives, users were frequently confronted with the question, which approach would be the most suitable going forward. This is why, in the following article, we want to describe several specific mechanisms that are crucial for the future in order to eliminate uncertainties and present the optimal course of action to developers.

Some of the previously available alternatives could not be supported anymore since it was not feasible for our software to consider severely outdated PC hardware without impeding technological advances at the same time. With the release of the security updates for the hardware bugs “Meltdown” and “Spectre” by Intel and Microsoft, some of the remaining solutions became restricted as well.

The following table shows what will be available for Kithara RealTime Suite starting with Version 10.05:

Alternative	Before	Now
Windows 7 or Windows 10?	both	both
Operating system: 32-bit or 64-bit?	both	both
Application: 32-bit or 64-bit?	both	both
Native or mixed size?	both	both
Windows interrupts or real-time interrupts?	both	real-time interrupts only (Exception: Interrupt Module)
Instruction-based relocation or kernel DLL?	both	via kernel DLL only
Shared real-time or dedicated real-time?	both	Dedicated real-time only
Standard PC or APIC available?	both	APIC has to be available
Single-core PC or multi-core PC?	both	multi-core PC

Apart from the required hardware equipment (multi-core PCs with APIC have been standard for years), the following will focus on the other restrictions. Windows 10 in general as well as all systems that had the security updates for counteracting the issues with “Meltdown” and “Spectre” installed, both have made shared real-time impossible. This means that, going forward, the procedure of running real-time tasks as well as Windows on the same CPU core will cease to function. The reason for this is the rigorous reduction and monitoring of the page-table-addressed system memory range which prevents the usage of real-time interrupts on CPUs that are operated by Windows.

This change has the consequence that running the dedicated real-time mode, meaning the utilization of logical CPUs exclusively for the real-time system, becomes mandatory. Excluding single CPUs can be done with the Windows tool “msconfig.exe”. In that case, the designated CPU will only boot the Kithara real-time system. What are “logical CPUs”? The term is supposed to lead away from definitions such as “cores” or “hyperthreads” and instead describe any kind of procedure that implements parallel instruction execution. Please see the following website on how to set up dedicated real-time mode for the Kithara real-time system (registration/login required):

> **http://kithara.com/en/docs/krts:tutorial:setupdedicated**

Here are the most important benefits for using dedicated real-time:

- Disabling of interrupts (CLI/STI) by external code is prevented
- Long instruction delays (e. g. WBINVD) by external code are prevented
- CPU cannot enter deep sleep states anymore (due to deactivation of C1E)
- No forced cyclical control release to Windows (allows for permanent use of CPU)
- Significantly higher interrupt frequencies achievable
- Significantly faster interrupt reaction as well as shorter task switching
- Proprietary system environment allows for optimized processes
- Optimization of interrupt concept
- Flexible hyperthreading application for every individual core
- Strong memory protection against accesses by external programs

In short: Real-time properties and safety mechanisms are significantly improved.

Essential components of a real-time application

For further consideration of the software architecture, the four major components that are involved in the execution of a real-time application need to be outlined:

Component	Type	Description
❶ Real-time kernel by Kithara	SYS	One-time installation as generic Windows driver
❷ Interface driver by Kithara	DLL	Provides user with access to the real-time kernel
❸ Windows application	EXE	Windows program of the user
❹ Kernel DLL	DLL	Real-time code of the user, loaded into real-time context

❶ The real-time kernel is a crucial program part of the real-time system. It specifies the organization of accessed resources, registering of handlers for the different events and also includes the implementation of almost all function modules.

In the course of the so-called runtime installation, the real-time system is installed like a device driver, in this case of a generic kind, and which supports a variety of specific hardware devices. However, it remains completely passive in the background until the first application establishes access to the real-time kernel.

❷ The interface driver is a DLL that provides access to the real-time kernel via exported API functions. Header files and import libraries for the different programming languages and development environments that are employed by the user are available.

❸ The user's Windows application can principally be generated in any programming language that is able to execute exported API functions from a DLL. Via the interface driver, the application gains access to the real-time kernel, generates and initializes necessary resources (such as shared memory, pipes, sockets or real-time tasks) and loads one or more kernel DLLs into the real-time context.

❹ The kernel DLL, on the other hand, contains the time-sensitive and hardware-dependent program code of the implementation, which is loaded into the real-time context by the Windows application after the necessary resources have been generated. By this point, it has access to all real-time functions of Kithara RealTime Suite. The kernel DLL has to be created with a programming language that allows for the generation of native machine code such as C/C++ or Delphi.

Basic differences between systems

The real-time extension Kithara RealTime Suite supports Windows 7 and Windows 10, each in both the 32-bit version as well as the 64-bit version of the operating systems. Regarding 64-bit Windows systems, it is further differentiated whether the Windows application exists as 32-bit version or 64-bit version. The differences between Windows 7 and Windows 10 are not relevant to this specific matter and can be safely ignored. This basically leads to three different variants.

Windows Operating System	Real-Time Kernel by Kithara	Kernel DLL	Windows Application	Explanation
❶ 32-bit	SYS: 32-bit	DLL: 32-bit	EXE: 32-bit	native
❷ 64-bit	SYS: 64-bit	DLL: 64-bit	EXE: 64-bit	native
❸ 64-bit	SYS: 64-bit	DLL: 64-bit	EXE: 32-bit	Mixed-Size: KiK64

While the first two variants seem plausible, since all components that are involved have the same bit size (32-bit or 64-bit), the third variant presents a special case, to which we will go into detail at a later point.

Programming languages and models

Kithara RealTime Suite users create a Windows application as user mode applications program and move the time-sensitive and hardware-dependent code parts into a DLL, which is then loaded into the real-time context by the driver. The following table shows how Windows application and kernel DLL respectively are able to gain access to functions of the real-time driver.

Programming Language/Model	Windows Application	Kernel DLL
C functions	✓	✓
Delphi functions	✓	✓
C# or other .NET languages (no native machine code)	✓	–

C/C++ as well as Delphi can load the interface driver either statically via an import library or, if required, integrate it dynamically. The Kithara interface driver provides access to real-time functionalities via C API functions, which is a common standard used with Windows operating systems to establish access from different programming languages and applications. All C functions return a value which either signifies the result or the error cause. The error code should generally be evaluated as it often leads to the source of the error.

Error handling in C (example):

```
const char* pCustomerNumber = "012345"; // Customer number at Kithara
if (KSError error = KS_openDriver(pCustomerNumber)) {
  // error handling, such as:
  outputErr(error, "KS_openDriver", "Unable to open the driver!");
  KS_closeDriver();
  return;
}
```

General approach

The user generally creates two components, the Windows application and the kernel DLL. However, since both are executed in different contexts (Windows context and real-time context), the question then arises how these two parts are able to exchange information with each other. There are indirect methods such as pipes or sockets but also shared memory for direct data exchange, which is a special memory range, that both sides have simultaneous access to.

The following example shows this procedure. It will describe how to set up a real-time timer with which to record measurement data in real-time and how to enable transmission to the Windows application via a data pipe.

Data exchange between Windows application and kernel DLL

For the flexible and typesafe composing of shared memory, we recommend defining a data structure (struct in C/C++ or record in Delphi) which includes all individual variables. In variants 1 and 2 in the table on page 4, meaning cases where all involved components possess the same bit size, type safety

is ensured automatically. It is only variant 3 that requires special attention, which we will cover at a later point.

Defining a structure for data exchange between Windows application and kernel DLL:

```
struct SharedData {
  // contains handles for data exchange between Windows application and
  // kernel DLL such as objects for events, pipes, tasks, network ports etc.

  KSHandle hKernel;
  KSHandle hEvent;
  KSHandle hPipe;
  KSHandle hCallBack;
  KSHandle hTask;
  KSHandle hTimer;
};
```

What happens in the Windows application?

The following code excerpts (in C/C++) explain how the Windows application generates different resources and how the kernel DLL is loaded.

Only after successfully opening the real-time driver, further functions are available:

```
const char* pCustomerNumber = "012345"; // Customer number at Kithara
if (KSError error = KS_openDriver(pCustomerNumber))
  ...
```

Creating a shared memory range for data exchange:

```
KSHandle hData;
if (KSError error = KS_createSharedMemEx(&hData, "SharedMemName",
    sizeof(SharedData), 0))
  ...
```

Determining the address of the shared memory range in the Windows context:

```
SharedData* pApp;
if (KSError error = KS_getSharedMemEx(hData, (void**) &pApp, 0))
  ...
```

Loading a kernel DLL into the real-time context:

```
if (KSError error = KS_loadKernel(&pApp->hKernel, "C:\\MyKernel.dll", NULL,
    NULL, 0))
  ...
```

Executing the initialization function *_initFunc* within the kernel DLL; it needs to receive the handle to the shared memory range:

```
if (KSError error = KS_execKernelFunction(pApp->hKernel, “initFunc”, &hData,
    NULL, KSF_NO_CONTEXT))
  ...
```

What happens in the kernel DLL?

First, the executed initialization function *_initFunc* in the kernel DLL also needs to determine the valid address for the common shared memory range of the real-time context (*pSys*). This should be stored as a global variable in the kernel DLL.

Prototype of the initialization function *_initFunc*:

```
extern "C"
KSError __declspec(dllexport) __stdcall _initFunc(void* pArgs) {
  KSHandle hData = *(KSHandle*) pArgs;
  ...
```

Determining the address of the shared memory range in the real-time context:

```
SharedData* pSys;
if (KSError error = KS_getSharedMemEx(hData, (void**) &pSys, 0))
  ...
```

Subsequently, all required resources are generated and the respective handles are stored in the common data structure.

Generating different resources (data pipe for the transmission of measurement data, event object for messaging the Windows application):

```
if (KSError error = KS_createEvent(&pSys->hEvent, "EventName", 0))
  ...
if (KSError error = KS_createPipe(&pSys->hPipe, "PipeName",
    sizeof(MeasurementData), 100, NULL, 0))
  ...
```

Generating a callback object from the exported function *_timerCallBack* of the kernel DLL for creating a real-time task with priority 200 and starting the task as real-time timer with a frequency of 10 kHz:

```
if (KSError error = KS_createKernelCallBack(&pSys->hCallBack, pSys->hKernel,
    "_timerCallBack", NULL, KSF_DIRECT_EXEC, 0))
  ...
if (KSError error = KS_createTask(&pSys->hTask, pSys->hCallBack, 200, 0))
  ...
if (KSError error = KS_createTimer(&pSys->hTimer, 100*us, pSys->hTask,
    KSF_REALTIME_EXEC))
  ...
```

What does the real-time timer task do within the kernel DLL?

The real-time task will be signaled after every sequence of the timer period, here meaning with a frequency of 10 kHz.

Callback function for the real-time task *_timerCallBack*:

```
extern "C"
KSError __declspec(dllexport) __stdcall _timerCallBack(void* pArgs,
    void* pContext) {
  SharedData* pData = (SharedData*) pArgs;
  TimerUserContext* pUserContext = (TimerUserContext*) pContext;
  ...
```

Writing measurement data into a data pipe and setting an event for notifying the Windows application of the presence of measurement data:

```
MeasurementData data;
... // 'data' mit Messdaten füllen
if (KSError error = KS_putPipe(pSys->hPipe, &data, 1, NULL, 0))
  ...
if (KSError error = KS_setEvent(pSys->hEvent))
  ...
```

Exiting the real-time task function (every return value unequal to 0 leads to cancellation):

```
..
  return KS_OK;
}
```

How does the Windows application receive the measurement data?

A Windows thread generated for the reception of measurement data is blocked by having the thread wait for the setting of the event object. After the event is set, the thread fetches the now arrived measurement data from the pipe and is then able to process, save or graphically visualize it:

Blocking a Windows thread by waiting for the event object:

```
if (KSError error = KS_waitForEvent(pApp->hEvent, 0, 0))
  ...
```

Reading measurement data from a data pipe:

```
MeasurementData data;
if (KSError error = KS_getPipe(pApp->hPipe, &data, 1, NULL, 0))
  ...
... // evaluate measurement data in 'data'
```

Summary

As shown, both components (Windows application as well as kernel DLL) are able to access the common memory range in order to exchange information.

In case it becomes necessary to use significantly larger memory blocks that are beyond this general method of data exchange, these can be generated appropriately in shared memory, whereas the handle is written into the already existing central structure of the type *SharedData*.

Special approach with KiK64

Until now, we only looked at the normal case, where all four components, meaning Windows operating system, real-time kernel by Kithara, Windows application as well as kernel DLL, are present with the same bit size, uniformly in 32-bit or 64-bit (variants 1 and 2 of the table on page 4). This poses a problem for many users when both 32-bit as well as 64-bit destination systems need to be supported, however, providing two Windows applications would also turn out to be too expensive or too complicated. As an example, for historical reasons, an application component, for which no 64-bit version exists, might be indispensable.

The seemingly obvious solution of having a single 32-bit Windows application support even 64-bit Windows systems, however, requires special attention. Whereas 32-bit versions of regular application programs are generally supported on 64-bit Windows, this is usually not possible at kernel level (in real-time context). The solution for this issue is the new KiK64 method. The following will detail this special function.

Even though KiK64 (Kithara-32-in-Kithara-64) still has the same name as previous versions, its new implementation, however, is completely different. Before, due to continuous address conversions, this feature was bought dearly with heavy performance loss and other limitations. The new approach, on the other hand, presents a flexible solution with full CPU performance and optimal real-time behaviour. The solution is to provide the kernel DLL natively for the respective Windows versions in both 32-bit as well as 64-bit (variant 3 of the table). Due to the inherently required limitation to real-time code, this can usually be implemented without any problems. This means that, depending on the present Windows installation, the 32-bit Windows application loads a 32-bit or a 64-bit kernel DLL into the real-time context.

Determining the bit size of an operating system:

```
KSSystemInformation info;
info.structSize = sizeof(KSSystemInformation);

if (KSError error = KS_getSystemInformation(&info, 0))
  ...
if (info.isSys64Bit)
  ... // 64-bit Windows system: load 64-bit kernel DLL
else
  ... // 32-bit Windows system: load 32-bit kernel DLL
```

In this particular case, if a 32-bit Windows installation is present, all other components of the 32-bit version are applied as well. The Windows application loads the 32-bit kernel DLL in a “native” mode, as described above.

In case a 64-bit Windows installation is present (as well as the real-time kernel by Kithara as 64-bit version), the Windows application loads the 64-bit kernel DLL into the real-time context. Special attention needs to be devoted to the outlining of data structures, which are created for shared memory. The reason is, at this point, that the Windows application has a “32-bit view” on the shared memory range, while the kernel DLL has a “64-bit view” on it. This is of utmost importance, as it needs to be ensured that both sides refer to an identical memory image, meaning the same bit size of variables within the shared data structures. How is this achieved?

Provided that only handles of the Kithara real-time environment or variables of simple data types are used, 32-bit and 64-bit views are identical, since all handles generally have the same size of 32 bit. However, the moment address pointers are also defined in data structures, the viewpoint of both sides would become different.

Due to the fact address pointers are only valid either in the Windows application or in the kernel DLL, this should generally be avoided. If an additional memory block for data exchange becomes necessary, another handle in turn has to be saved in shared memory. The following code excerpts show this.

False: Defining a structure for data exchange, which has to include an address pointer, that is created by `KS_createSharedMem`:

```
struct SharedData {
  Handle hEvent;
  byte* pBigBuffer; // <== 32-bit? 64-bit?
  Handle hPipe;
  Handle hSocket;
};
```

Defining a structure this way would be faulty. At the point where the application registers the socket handle *hSocket*, the kernel DLL will "think" that there is the pipe handle *hPipe*. Additionally, the Windows application would not be able to request a 64-bit-conform address pointer with `KS_createSharedMem` for the creation of shared memory to save it to the data structure since it would only receive the lower 32-bit part of it.

Correct: Defining a structure for data exchange that receives a handle on another shared memory block, created with `KS_createSharedMemEx`:

```
struct SharedData {
  Handle hEvent;
  Handle hBigBuffer;
  Handle hPipe;
  Handle hSocket;
};
```

Retrieving the correct memory address within the respective context environment:

```
BigBufferData* pBigBuffer;
if (KSError error = KS_getSharedMemEx(pSys->hBigBuffer, (void**) &pBigBuffer, 0))
  ...
pBigBuffer->...
```

While this approach is generally correct, for efficiency concerns, it should only be done once at the start of program execution (in *_initFunc*). Rather, it would be preferable to save the retrieved address pointer for later access. To conclude, here is a check list for real-time applications on 32-bit and 64-bit Windows systems:

1. Relocation of time-sensitive and hardware-dependent real-time code into a DLL project
2. Compilation of the DLL project as 32-bit DLL as well as 64-bit DLL
3. Setting up dedicated real-time mode on a system for one or more CPUs
4. Creation of shared memory with the function `KS_createSharedMemEx`
5. Determining specifically whether a 32-bit system or a 64-bit system is at hand
6. Loading of the appropriate DLL with the function `KS_loadKernel`
7. Creation of callback objects with the function `KS_createKernelCallBack`
8. Saving all resources that need to be accessed simultaneously, preferably as handles
9. Saving all potentially required address pointers locally

Conclusion

The excellent real-time capabilities of the Kithara environment, which, in case of native execution of real-time code, have existed for years, are now also available even for customers who had to avoid porting their Windows application into the 64-bit world. Going forward, this flexible and easily applicable support for both 32-bit as well as 64-bit Windows systems will be included with all 64-bit versions of the software.

Relocating real-time code into a DLL and executing it on a dedicated real-time CPU has now become mandatory. Both aspects, however, allow for significantly more stable real-time performance and more efficient multitasking.

With the new KiK64 concept combined with the obligatory utilization of the dedicated mode, the software architecture of Kithara RealTime Suite is perfectly equipped for future progress of hardware and operating systems.

Kithara

Imprint

Kithara Software GmbH, Alte Jakobstr. 78, 10179 Berlin, Germany, Phone: +49 30 2789673-0, Fax: +49 30 2789673-20
Email: info@kithara.com (general enquiries), support@kithara.com (technical enquiries), Internet: www.kithara.com