White Paper

Zukunftsfähige Software-Architektur mit Kithara RealTime Suite

Mit dem Dedicated-Modus und KiK64 in die Zukunft

Kithara RealTime Suite ist eine Echtzeiterweiterung für Windows-Betriebssysteme. Es handelt sich dabei um eine modulare Software, die in den mehr als 20 Jahren ihres Bestehens rein äußerlich nur wenige Wandlungen durchlaufen hat. Wenn im Zuge der Weiterentwicklung zugrunde liegender Betriebssysteme und Hardware-Plattformen Anpassungen im Inneren notwendig waren, sollte dies größere Änderungen für die Anwender möglichst vermeiden, denn die Kompatibilität mit früheren Versionen der Software hatte immer einen hohen Stellenwert.

In der Folge wurden in verschiedener Hinsicht mehrere alternative Ansätze unterstützt, die Variantenvielfalt nahm stetig zu. Unter anderem musste Kithara RealTime Suite im Laufe der Zeit Unterscheidungen hinsichtlich Hardware, Betriebssystem, Bitbreite verschiedener Programmteile des Anwenders, Kontext der Interrupt-Behandlung sowie Art und Weise der Echtzeitausführung beachten. Aufgrund der Vielzahl an Alternativen standen die Anwender immer mehr vor der Frage, welcher Ansatz denn nun für künftige Entwicklungen der günstigste sei. Wir wollen daher im vorliegenden Artikel auf spezielle und für die Zukunft wichtige Mechanismen eingehen, um Unklarheiten zu beseitigen und Entwicklern die optimale Vorgehensweise näherzubringen.

Einige der bislang verfügbaren Alternativen wurden nicht mehr unterstützt, da unsere Software auf gänzlich veraltete PC-Technik keine Rücksicht mehr nehmen konnte, ohne gleichzeitig den Fortschritt zu behindern. Mit Veröffentlichung der Security-Updates für die Hardware-Bugs „Meltdown“ und „Spectre“ durch Intel und Microsoft wurden einige der verbliebenen Lösungsansätze weiter eingeschränkt.

Die folgende Tabelle zeigt auf, was mit Kithara RealTime Suite ab Version 10.05 auch künftig möglich ist:

Alternative	früher	künftig
Windows 7 oder Windows 10?	beides	beides
Betriebssystem: 32 Bit oder 64 Bit?	beides	beides
Anwendung: 32 Bit oder 64 Bit?	beides	beides
Nativ oder „Mixed-Size“?	beides	beides
Windows-Interrupts oder Echtzeit-Interrupts?	beides	nur Echtzeit-Interrupts (Ausnahme: Interrupt Modul)
Befehlsweise Relozierung oder Kernel-DLL?	beides	nur über Kernel-DLL
Shared Real-Time oder Dedicated Real-Time?	beides	nur Dedicated Real-Time
Standard-PC oder APIC vorhanden?	beides	APIC muss vorhanden sein
Single-Core-PC oder Multi-Core-PC?	beides	nur Multi-Core-PC

Abgesehen von der notwendigen Hardware-Ausstattung (Multi-Core-PC mit APIC sind seit vielen Jahren üblich) sollen hier die anderen Einschränkungen betrachtet werden. Sowohl Windows 10 generell, als auch sämtliche Systeme nach Einspielen der Security Updates zur Linderung der Problematik von „Meltdown“ und „Spectre“ machen Shared Real-Time, also die Nutzung eines CPU-Kerns sowohl für Echtzeitaufgaben als auch für Windows, künftig unmöglich. Der Grund ist eine rigorose Verkleinerung und Überwachung des Page-Table-adressierten Systemspeicherbereiches, der eine Nutzung von Echtzeit-Interrupts auf von Windows verwalteten CPUs verhindert.

Das bedeutet, es muss zwingend der Dedicated-Real-Time-Modus benutzt werden. Dahinter steckt die Verwendung einzelner logischer CPUs ausschließlich durch das Echtzeitsystem. Das Freihalten einzelner CPUs kann mit dem Windows-Tool „msconfig.exe“ vorgenommen werden. In dem Fall wird auf diesen das Kithara-Echtzeitsystem gebootet. Doch warum ist hier von „logischen CPUs“ die Rede? Dieser Begriff soll von „Cores“ oder „Hyperthreads“ wegführen und verallgemeinern, auf welche Weise auch immer eine parallele Befehlsausführung realisiert wird. Bitte entnehmen Sie der folgenden Webseite Hinweise zur Einrichtung des Dedicated-Real-Time-Modus für das Kithara-Echtzeitsystem (Registrierung/Login erforderlich):

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

Die wichtigsten Vorteile bei der Nutzung des Dedicated-Real-Time-Modus sind:

- Interrupt-Sperrungen (CLI/STI) durch Fremd-Code nicht möglich
- Lange Befehls-Delays (z. B. WBINVD) durch Fremd-Code werden verhindert
- CPU kann nicht mehr in zu tiefe Schlafzustände fallen (durch Abschaltung von C1E)
- Zyklische Kontrollabgabe an Windows nicht nötig (Dauerbenutzung der CPU möglich)
- Deutlich höhere Interrupt-Frequenzen möglich
- Wesentlich schnellere Interrupt-Annahme und kürzere Task-Wechselzeiten
- Eigene Systemumgebung erlaubt optimierte Abläufe
- Optimierung des Interrupt-Schemas
- Freie Entscheidung über die Nutzung von Hyperthreading je Kern
- Sehr starker Speicherschutz vor Zugriffen aus Fremdprogrammen

Kurzum: Die Echtzeiteigenschaften und die Schutzfunktionen verbessern sich enorm.

Wesentliche Komponenten einer Echtzeit-Applikation

Zur weiteren Betrachtung der Software-Architektur sollen die hauptsächlichen vier Komponenten dargestellt werden, die bei der Ausführung einer Echtzeitanwendung involviert sind:

Bestandteil	Typ	Beschreibung
❶ Echtzeit-Kernel von Kithara	SYS	einmalig als generischer Windows-Treiber installiert
❷ Interface-Treiber von Kithara	DLL	stellt den Zugang des Anwenders zum Echtzeit-Kernel bereit
❸ Windows-Applikation	EXE	Windows-Programm des Anwenders
❹ Kernel-DLL	DLL	Echtzeit-Code des Anwenders, in Echtzeitkontext geladen

❶ Der Echtzeit-Kernel ist ein wesentlicher Programmteil des Echtzeitsystems. In ihm werden Zugriffe auf Ressourcen organisiert, Handler für die verschiedenen Ereignisse angemeldet, und die Implementierung fast aller Funktionsmodule findet sich dort.

Das Echtzeitsystem wird im Zuge der sogenannten Runtime-Installation wie ein Gerätetreiber installiert, der in diesem Fall generischer Natur ist und verschiedene konkrete Hardware-Devices unterstützen kann. Er verhält sich jedoch völlig passiv im Hintergrund, bis die erste Anwendung den Zugang zum Echtzeit-Kernel herstellt.

❷ Der Interface-Treiber ist eine DLL, die über exportierte API-Funktionen den Zugang zum Echtzeit-Kernel bereitstellt. Header-Dateien und Importbibliotheken stehen für verschiedene Programmiersprachen und Entwicklungsumgebungen des Anwenders zur Verfügung.

❸ Die Windows-Applikation des Anwenders kann prinzipiell in jeder beliebigen Programmiersprache erstellt werden, mit der man in der Lage ist, von einer DLL exportierte API-Funktionen auszuführen. Sie stellt über den Interface-Treiber den Zugang zum Echtzeit-Kernel her, erzeugt anschließend einen Shared-Memory-Block für die gemeinsame Verwaltung benötigter Ressourcen und lädt eine oder mehrere Kernel-DLLs in den Echtzeitkontext.

❹ Die Kernel-DLL wiederum wird von der Windows-Applikation in den Echtzeitkontext geladen. Sie übernimmt das eigentliche Bereitstellen benötigter Ressourcen (wie zusätzliches Shared Memory, Events, Pipes, Sockets oder Echtzeit-Tasks). Darüber hinaus enthält sie den zeitkritischen oder hardwarenahen Programmcode der Anwendung. Die Kernel-DLL hat dann Zugriff auf sämtliche Echtzeitfunktionen von Kithara RealTime Suite. Sie muss mit einer Programmiersprache erstellt werden, die nativen Maschinencode erzeugen kann, wie etwa C/C++ oder Delphi.

Grundsätzliche Systemunterschiede

Die Echtzeiterweiterung Kithara RealTime Suite unterstützt Windows 7 und Windows 10, jeweils die 32- und 64-Bit-Version der Betriebssysteme. Zusätzlich wird unterschieden, ob auf einem 64-Bit-Windows-System die Windows-Applikation des Anwenders als 32- oder als 64-Bit-Variante vorliegt. Die Unterschiede zwischen Windows 7 und Windows 10 sind hierbei nicht relevant und können daher vernachlässigt werden. Es ergeben sich also im Wesentlichen zunächst drei unterschiedliche Varianten.

Windows-Betriebssystem	Echtzeit-Kernel von Kithara	Kernel-DLL	Windows-Applikation	Erläuterung
❶ 32 Bit	SYS: 32 Bit	DLL: 32 Bit	EXE: 32 Bit	nativ
❷ 64 Bit	SYS: 64 Bit	DLL: 64 Bit	EXE: 64 Bit	nativ
❸ 64 Bit	SYS: 64 Bit	DLL: 64 Bit	EXE: 32 Bit	Mixed-Size: KiK64

Während die ersten beiden Varianten sehr eindeutig erscheinen, da alle beteiligten Komponenten die gleiche Bitbreite (32 oder 64 Bit) besitzen, stellt die dritte Variante eine Besonderheit dar, auf die später noch eingegangen wird.

Programmiersprachen und -modelle

Anwender von Kithara RealTime Suite erstellen eine Windows-Applikation als User-Mode-Anwendungsprogramm und lagern die zeitkritischen und hardwarenahen Code-Teile in eine DLL aus, die dann vom Treiber in den Echtzeitkontext geladen wird. Die folgende Tabelle verdeutlicht, wie Win-

dows-Applikation beziehungsweise Kernel-DLL Zugang zu den Funktionen des Echtzeittreibers erhalten können.

Programmiersprache/-modell	Windows-Applikation	Kernel-DLL
C-Funktionen	✓	✓
Delphi-Funktionen	✓	✓
C# oder andere .NET-Sprachen (kein nativer Maschinencode)	✓	–

Sowohl C/C++ als auch Delphi können den Interface-Treiber entweder statisch über eine Importbibliothek laden oder nur bei Bedarf dynamisch einbinden. Der Interface-Treiber von Kithara stellt den Zugang zur Echtzeitfunktionalität über C-API-Funktionen bereit. Dies ist ein bei Windows-Betriebssystemen verwendeter Standard für den Zugang aus unterschiedlichen Applikationen und Programmiersprachen. Alle C-Funktionen liefern einen Wert zurück, der den Erfolg oder die Fehlerursache anzeigt. Der Fehlercode sollte generell ausgewertet werden. Im Falle eines Fehlers lässt sich dadurch meistens auf die Fehlerursache schließen.

Fehlerbehandlung in C (Beispiel):

```
const char* pCustomerNumber = "012345"; // Kundennummer bei Kithara
if (KSError error = KS_openDriver(pCustomerNumber)) {
  // Fehlerbehandlung, z.B.:
  outputErr(error, "KS_openDriver", "Unable to open the driver!");
  KS_closeDriver();
  return;
}
```

Allgemeine Vorgehensweise

Generell erstellt der Anwender zwei Komponenten – die Windows-Applikation und die Kernel-DLL. Da beide jedoch in unterschiedlichen Kontexten ausgeführt werden (Windows-Kontext beziehungsweise Echtzeitkontext), stellt sich die Frage, wie diese beiden Teile miteinander Informationen austauschen können. Dazu stehen einerseits indirekte Mittel, wie Pipes oder Sockets, andererseits aber auch Shared Memory für den direkten Datenaustausch zur Verfügung. Dabei handelt es sich um einen speziellen Speicherbereich, auf den beide Seiten gleichzeitig Zugriff haben.

In dem folgenden Beispiel wird die Vorgehensweise verdeutlicht. Es soll ein Echtzeit-Timer eingerichtet werden, in dem Messdaten in Echtzeit erfasst und mittels einer Datenpipe an die Windows-Applikation gegeben werden.

Datenaustausch zwischen Windows-Applikation und Kernel-DLL

Zum flexiblen und typsicheren Entwurf des Shared Memory empfehlen wir, eine Datenstruktur zu definieren (struct in C/C++ beziehungsweise record in Delphi), die alle einzelnen Variablen enthält. Die Typsicherheit ist bei den Varianten 1 und 2 der auf Seite 4 abgebildeten Tabelle, also wenn alle beteiligten Komponenten die gleiche Bitgröße besitzen, automatisch sichergestellt. Besonders zu beachten ist nur die Variante 3, auf die später eingegangen wird.

Definition einer Struktur zum Datenaustausch zwischen Windows-Applikation und Kernel-DLL:

```
struct SharedData {
  // enthält Handles für Datenaustausch zwischen Windows-Applikation und
  // Kernel-DLL, z.B. Objekte für Events, Pipes, Tasks, Netzwerk-Ports etc.:

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

Was geschieht in der Windows-Applikation?

Die folgenden Code-Auszüge (in C/C++) beschreiben, wie die Windows-Applikation verschiedene Ressourcen erzeugt und die Kernel-DLL geladen wird.

Erst nach dem erfolgreichen Öffnen des Echtzeittreibers sind die weiteren Funktionen ausführbar:

```
const char* pCustomerNumber = "012345"; // Kundennummer bei Kithara
if (KSError error = KS_openDriver(pCustomerNumber))
  ...
```

Erzeugen eines Shared-Memory-Bereiches für den Datenaustausch:

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

Ermitteln der Adresse des Shared-Memory-Bereiches im Windows-Kontext:

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

Laden der Kernel-DLL in den Echtzeitkontext:

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

Ausführen einer Initialisierungsfunktion *_initFunc* in der Kernel-DLL; dieser muss das Handle auf den Shared-Memory-Bereich übermittelt werden:

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

Was geschieht in der Kernel-DLL?

Die damit zur Ausführung gebrachte Initialisierungsfunktion *_initFunc* in der Kernel-DLL muss nun ebenfalls zunächst die im Echtzeitkontext gültige Adresse des gemeinsamen Shared-Memory-Bereiches ermitteln (*pSys*). Diese sollte als globale Variable in der Kernel-DLL abgelegt werden.

Prototyp der Initialisierungsfunktion *_initFunc*:

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

Ermitteln der Adresse des Shared-Memory-Bereiches im Echtzeitkontext:

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

Anschließend werden sämtliche benötigten Ressourcen erzeugt und die jeweiligen Handles in der gemeinsamen Datenstruktur hinterlegt.

Erzeugen verschiedener Ressourcen (Datenpipe für die Übermittlung der Messdaten, Event-Objekt zur Mitteilung an die Windows-Applikation):

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

Erzeugen eines Callback-Objektes der Kernel-DLL aus der exportierten Funktion *_timerCallBack* für einen Echtzeit-Task mit der Priorität 200 und Anmeldung des Tasks als Echtzeit-Timer mit einer Frequenz von 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))
  ...
```

Was macht der Echtzeit-Timer-Task in der Kernel-DLL?

Der Echtzeit-Task wird nach jedem Ablauf der Timer-Periode (hier also mit einer Frequenz von 10 kHz) signalisiert.

Callback-Funktion für den Echtzeit-Task *_timerCallBack*:

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

Schreiben von Messdaten in eine Datenpipe und Setzen eines Events, um der Windows-Applikation das Vorliegen von Messdaten zu signalisieren:

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

Verlassen der Echtzeit-Task-Funktion (jeder Rückgabewert ungleich 0 bedeutet Abbruch):

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

Wie erwartet die Windows-Applikation die Messdaten?

Ein für die Entgegennahme der Messdaten erzeugter Windows-Thread wird blockiert, indem er auf das Setzen des Event-Objektes wartet. Nach dem Setzen des Events holt der Thread die inzwischen eingetroffenen Messdaten aus der Pipe ab und kann diese weiterverarbeiten, speichern oder grafisch darstellen.

Blockieren des Windows-Threads durch Warten auf das Event-Objekt:

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

Lesen der Messdaten aus der Datenpipe:

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

Zusammenfassung

Wie ersichtlich ist, verfügen beide Komponenten (Windows-Applikation sowie Kernel-DLL) über die Möglichkeit, auf den gemeinsamen Speicherbereich zuzugreifen, um hierüber Informationen auszutauschen.

Sollte es erforderlich sein, neben diesem zentralen Mechanismus zum Datenaustausch weitere, eventuell deutlich größere Speicherblöcke zu verwenden, lassen sich diese wiederum auf geeignete Weise im Shared Memory erzeugen, während deren Handles in die bereits vorhandene zentrale Struktur vom Typ *SharedData* geschrieben werden.

Spezielle Vorgehensweise mit KiK64

Bisher wurde nur der Normalfall betrachtet, nämlich dass alle vier Komponenten, also Windows-Betriebssystem, Echtzeit-Kernel von Kithara, Windows-Applikation und Kernel-DLL, in der gleichen Bitgröße vorliegen, also entweder einheitlich 32 Bit oder 64 Bit (Varianten 1 und 2 der auf Seite 4 dargestellten Tabelle). Für viele Anwender stellt es ein Problem dar, wenn sowohl 32- als auch 64-Bit-Zielsysteme unterstützt werden sollen, die Bereitstellung zweier Windows-Applikationen jedoch zu aufwändig oder problematisch ist. Es könnte zum Beispiel auch historisch bedingt eine Anwendungskomponente verwendet worden sein, für die es keine 64-Bit-Fassung gibt.

Die naheliegende Lösung, mit einer einzigen 32-Bit-Windows-Applikation auch 64-Bit-Windows-Systeme zu unterstützen, erfordert jedoch besondere Aufmerksamkeit: Während normale Anwendungen auch als 32-Bit-Version auf einem 64-Bit-Windows betrieben werden können, ist dies auf der Kernel-Ebene (im Echtzeitkontext) nicht möglich. Die Lösung hierfür bietet die neue KiK64-Unterstützung. Im Folgenden wird auf diese spezielle Funktion eingegangen.

KiK64 (Kithara-32-in-Kithara-64) trägt zwar noch die gleiche Bezeichnung wie frühere Versionen, wird jedoch gänzlich anders umgesetzt. Was bisher aufgrund fortdauernder Adressumrechnungen nur mit Leistungseinbußen und anderen Einschränkungen erkauft werden konnte, bietet in Zukunft eine flexible Möglichkeit mit voller CPU-Leistung und optimalem Echtzeitverhalten. Die Lösung besteht darin, dass die Kernel-DLL nativ, sowohl in 32 als auch 64 Bit, für die in Frage kommenden Windows-Versionen bereitgestellt wird (Variante 3 der Tabelle). Dies ist durch die ohnehin erforderliche Beschränkung auf Echtzeit-Code in der Regel problemlos umsetzbar. Die 32-Bit-Windows-Applikation lädt also je nach vorliegender Windows-Installation eine 32-Bit- oder eine 64-Bit-Kernel-DLL in den Echtzeitkontext.

Feststellen der Bitgröße des Betriebssystems:

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

if (KSError error = KS_getSystemInformation(&info, 0))
  ...
if (info.isSys64Bit)
  ... // 64-Bit-Windows-System: 64-Bit-Kernel-DLL laden
else
  ... // 32-Bit-Windows-System: 32-Bit-Kernel-DLL laden
```

Handelt es sich nun im konkreten Fall um eine 32-Bit-Windows-Installation, so werden auch alle anderen Komponenten als 32-Bit-Fassung verwendet. Die Windows-Applikation lädt die 32-Bit-Kernel-DLL nach „nativem“ Konzept, wie oben beschrieben.

Falls es sich jedoch um eine 64-Bit-Windows-Installation handelt (und der Echtzeit-Kernel von Kithara liegt damit auch als 64-Bit-Version vor), so lädt die Windows-Applikation nun die 64-Bit-Kernel-DLL in den Echtzeitkontext. Dabei ist dem Entwurf aller Datenstrukturen, für die Shared Memory erzeugt werden soll, besondere Aufmerksamkeit zu widmen. Der Grund ist, dass nun die Windows-Applikation eine „32-Bit-Sicht“, die Kernel-DLL jedoch eine „64-Bit-Sicht“ auf den gemeinsamen Speicher hat. Dies spielt eine entscheidende Rolle, da sichergestellt werden muss, dass für beide Seiten das Speicherabbild, also die Bitgröße der Variablen in den gemeinsamen Datenstrukturen, identisch aussieht. Wie erreicht man das?

Sofern ausschließlich Handles der Kithara-Echtzeitumgebung oder Variablen einfacher Datentypen verwendet werden, sind die 32-Bit-Sicht und die 64-Bit-Sicht identisch, denn wie bereits weiter oben erläutert, besitzen alle Handles generell eine Größe von 32 Bit. Sobald jedoch auch Adresszeiger in der Datenstruktur definiert werden, würde sich die Sichtweise der beiden Seiten unterscheiden.

Da Adresszeiger jedoch entweder in der Windows-Applikation oder in der Kernel-DLL gültig sind, sollte dies generell vermieden werden. Bei Bedarf nach einem weiteren Speicherblock zum Datenaustausch ist wiederum ein Handle auf Shared Memory abzulegen. Folgende Code-Auszüge zeigen dies.

Falsch: Definition einer Struktur zum Datenaustausch, die einen Adresszeiger enthalten soll, der mit `KS_createSharedMem` erzeugt wird:

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

Eine solche Definition der Struktur wäre fehlerhaft. An der Stelle, an der die Applikation das Socket-Handle *hSocket* eingetragen hat, „meint" die Kernel-DLL das Pipe-Handle *hPipe* vorzufinden. Abgesehen davon wäre die Windows-Applikation nicht in der Lage, einen in der 64-Bit-Welt gültigen Adresszeiger mit `KS_createSharedMem` zum Erzeugen von Shared Memory abzufordern und in der Datenstruktur zu speichern, da sie nur den niederwertigen 32-Bit-Teil davon bekäme.

Richtig: Definition einer Struktur zum Datenaustausch, die ein Handle auf einen weiteren Shared-Memory-Block erhält, der mit `KS_createSharedMemEx` erzeugt wird:

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

Abrufen der konkreten Adresse des Speichers in der jeweiligen Kontextumgebung:

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

Dieser Ansatz ist zunächst formal richtig, sollte jedoch aus Effizienzgründen nur einmalig zu Beginn der Programmausführung erfolgen (in *_initFunc*). Besser ist es, den erhaltenen Adresszeiger für spätere Zugriffe zu speichern. Abschließend folgt noch einmal eine Checkliste für Echtzeitanwendungen auf 32- und 64-Bit-Windows-Systemen:

1. Verlagern des zeitkritischen und hardwarenahen Echtzeit-Codes in ein DLL-Projekt
2. Kompilieren des DLL-Projektes als 32- sowie als 64-Bit-DLL
3. Einrichten des Dedicated-Echtzeit-Modus‘ auf dem System für eine oder mehrere CPUs
4. Erzeugen von Shared Memory mit der Funktion `KS_createSharedMemEx`
5. Feststellen, ob es sich um ein 32- oder 64-Bit-System handelt
6. Laden der jeweils passenden DLL mit der Funktion `KS_loadKernel`
7. Erzeugen der Callback-Objekte mit der Funktion `KS_createKernelCallBack`
8. Speichern aller gemeinsam anzusprechenden Ressourcen als Handles
9. Lokales Speichern eventuell erforderlicher Adresszeiger

Fazit

Die hervorragenden Echtzeiteigenschaften der Kithara-Umgebung, die bei nativer Ausführung von Echtzeit-Code seit Jahren gegeben waren, stehen nun auch jenen Kunden offen, welche sich die Portierung ihrer Windows-Applikation in die 64-Bit-Welt ersparen wollen oder müssen. Diese flexible und einfach umsetzbare Unterstützung von sowohl 32- als auch 64-Bit-Windows-Systemen ist ab sofort bei allen 64-Bit-Varianten der Software mit inbegriffen.

Um die Auslagerung des Echtzeit-Codes in eine DLL und die Ausführung auf einer dedizierten Echtzeit-CPU kommt künftig niemand mehr herum. Beide Punkte ermöglichen künftig jedoch eine stabilere Echtzeitleistung sowie effizienteres Multitasking.

Sowohl durch den neuen KiK64-Ansatz als auch den verbindlichen Einsatz des Dedicated-Modus‘ ist die Software-Architektur mit Kithara RealTime Suite rundum für zukünftige Entwicklungen der Hardware und Betriebssysteme bestens gerüstet.

Kithara

Impressum

Kithara Software GmbH, Alte Jakobstraße 78, 10179 Berlin, Germany, Telefon: +49 30 2789673-0, Fax: +49 30 2789673-20
E-Mail: info@kithara.com (allgemeine Anfragen), support@kithara.com (technische Anfragen), Internet: www.kithara.com