Dr. E.Huckert 12-2010

Design von Echtzeitanwendungen

Table of Contents

Überblick

Der folgende Artikel erläutert zunächst einige mir wichtige Prinzipien und Ziele im Entwurf von Echtzeitanwendungen. Daran anschließend wird ein in C++ geschriebenes Programm erläutert, in dem die beschriebenen Prinzipien umgesetzt sind.

Abgrenzung

Es gibt keine tragfähige Definition für den Begriff „Echtzeitsystem". Einig ist man sich nur darin, dass ein typisches Echtzeitsystem je nach Anwendungszweck minimale Reaktionszeiten garantieren muss.

Es gibt auch keine „typische" Echtzeitanwendung. Oft wird angenommen, dass eine Echtzeitanwendung irgendeine Form von **Interrupt-Serviceroutine** enthält. In modernen Betriebsystemen findet man so etwas fast nur noch in Treibern, also eher auf der Ebene des Betriebssystems. Auch die Implementierung von **Treibern** – hardwarenahe Programme – ist nicht unbedingt typisch für eine Echzeitanwendung. Interrupt-Serviceroutinen und Treiber können als eine Art von Tasks angesehen werden, die nicht den üblichen Start- und Terminierungsmechanismen unterliegen. Sie verwenden oftmals die gleichen Synchronisations- und Kommunikationsmechanismen wie normale Tasks in Echtzeitanwendungen.

Echtzeitsystemen müssen nicht unbedingt auf reinen Echtzeit - Betriebssystemen aufsetzen. Wenn zusätzlich angenommen wird, dass nur ein Benutzer auf dem System angemeldet ist und dass keine unnötigen Hintergrundprozesse laufen, können Echtzeitanwendungen auch mit neueren konventionellen Betriebssystemen wie Unix/Linux oder Windows aufgesetzt werden, vorausgesetzt, diese stellen die unten beschrieben Kommunikationsmechanismen zur Verfügung.

Echtzeitsysteme verwenden meist irgendeine Form paralleler Prozesse oder paralleler Threads oder gar eine Mischung von Threads und Prozessen. Wir verwenden hier parallele Threads, die etwas vereinfacht auch „Tasks" genannt werden.

Wir vernachlässigen hier die Probleme, die durch den Ablauf mehrerer Tasks auf parallelen CPUs entstehen. Diese Variante von Echtzeitsystemen bringt zusätzliche Probleme mit sich, die unseren Rahmen überschreiten. Wir vernachlässigen auch das „verteilte Rechnen auf Systemen mit verteiltem Speicher" (s. Bauke/Mertens[2006]), das nichts nit den hier besprochenen Problemen und Lösungen zu tun hat.

Entwurfsziele in Echtzeitanwendungen

Einfachheit

Nach Jahren der Entwicklung von Echtzeitanwendungen und normalen kommerziellen Anwendungen komme ich zur Überzeugung: **es gibt nur ein übergeordnetes Entwurfsziel für Echtzeitanwendungen: das der Einfachheit.**

Einfachheit lässt sich nicht ganz einfach definieren – sie ist eher intuitiv erfassbar. Nur einige Kriterien sollten genannt werden:

- der Systementwurf (das „Bild") sollte intuitiv verständlich sein
- die Zahl der Systemkomponenten (Tasks, Prozesse, Schnittstellen) sollte so gering wie möglich und sinnvoll sein
- die Zahl der verwendeten Softwarekomponenten (Module, Bibliotheken) sollte so gering und homogen (keine Mischung von Sprachen oder Herstellern!) wie möglich sein
- keine Techniken verwenden, nur weil sie in aller Munde sind. XML z.B. ist nur sehr selten wirklich sinnvoll. Wenn nur ab und zu einige Daten in eine Datenbank geschrieben werden sollen, ist ein objektrelationales Mapping nicht unbedingt erforderlich.
- Keine Mischung von typischen Echtzeittechniken: wenn z.B. Sperren von Ressourcen nötigt ist, dann nicht Mutexe, Semaphore, „timed conditions" etc. mischen. Eine einzige Technik genügt.
- Die besten Echtzeittechniken sind die, auf die man verzichten kann. So ist es oft möglich, auf Sperrmechanismen für Codestrecken in Prozeduren komplett zu verzichten, wenn Tasks sauber entkoppelt sind (s. unten).
- keine Mischung von Entwurfstechniken: also nicht z.B. objektorientierte Ansätze mit rein funktionalen Ansätzen mischen. Es gibt per se keinen Entwurfsansatz, der besser als ein anderer wäre. Es gibt allerdings viele inkonsequent verfolgte Ansätze, die zudem mit völlig konträren Ansätzen gemischt werden.

Die Einfachheit eines Systems („small is beautiful", KISS Prinzip) wird in vielen Situationen belohnt:

- die Anwendung kann einfacher fehlerbereinigt werden, wenn Fehler auftreten – weil es durchschaubarer ist
- bei Leistungsengpässen können Tuningmaßnahmen leichter geplant und durchgeführt werden
- die Dokumentation des System ist weniger umfänglich
- die Wartbarkeit ist erhöht

Effizienz

Effizienz in Echtzeitsystemen bedeutet normalerweise: die Anwendung muss die vorgegebenen Mindestanforderungen (z.B. die Reaktionszeiten) einhalten und dabei auch noch Reserven haben.

Effizienz aus Entwicklersicht bedeutet aber auch: möglichst schnell zum Entwurfsziel – einer lauffähigen Anwendung zu kommen. Aus Managementsicht heißt das natürlich auch: kostengünstig zum Ziel kommen, möglichst wenig Ressourcen verbrauchen.

Im Sinne der Einfachheit bedeutet Effizienz vor allem: zur Laufzeit möglichst wenige Betriebsmittel verbrauchen und zur Entwurfszeit möglichst wenige Komponenten verwenden. Die Planung der erreichbaren Effizienz – z.B. der voraussichtlichen Reaktionszeit – wird stark vereinfacht, wenn wenige Mechanismen zum Einsatz kommen und wenn das Verhalten dieser wenigen Mechanismen gut bekannt ist.

Debugfähigkeit

Debuggen in Echtzeitanwendungen ist ein besonderes Thema. Nicht jeder Wald-und-Wiesen Debugger ist für das Testen von Echtzeitanwendungen geeignet. Echtzeitanwendungen verhalten sich komplett anders als die üblichen kommerziellen Anwendungen, weil ihr Verhalten zeitlich selten vorhersagbar ist. So kann z.B. schon das Kompilieren im Debug-Modus zu einem geänderten Zeitverhalten führen. Beim Einschalten der Debug-Option des Compilers wird meist automatisch die Optimierung ausgeschaltet.

Sobald der Debugger auf einen Breakpoint läuft und das Programm damit unterbrochen wird, ändert sich das Zeitverhalten komplett. Im Hintergrund laufen ja Prozesse weiter – etwa eine Maschinensteuerung die nicht einfach angehalten werden darf. Damit wird auch die Reihenfolge der Ereignisse verändert, die eventuell zu einem Problem führen können. Die kritischen Probleme treten verstärkt oder gar nicht auf.

Meine These ist: Debugger sollten in Echtzeitanwendungen nie das erste Mittel zur Fehlersuche sein. Statt dessen verwende ich eher Logging – und dabei bevorzugt Logging auf UDP Basis. Logging verändert zwar auch das Zeitverhalten von Echtzeitanwendungen. Bei geschicktem Einsatz von Logging kann jedoch das Zeitverhalten uniform über die gesamte Anwendung verändert werden sodass dann wieder nachvollziehbare Verhältnisse auftreten.

Warum UDP basiertes Logging und nicht Logging in Dateien? Ganz einfach: viele kleine echtzeitfähige Systeme verfügen über kein Dateisystem oder haben nur sehr eingeschränkte Plattenkapzitäten, die schnell erschöpft sind. Logging über UDP (UDP blockiert anders als TCP keinen Netzwerkkanal) verlagert das Aufzeichnungsproblem auf einen externen Rechner, der hoffentlich über genügend Plattenkapazität verfügt. In sehr frühen Echtzeitsystemen war – ähnlich dem UDP Logging – aus den gleichen Gründen das Logging über die serielle Schnittstelle üblich und sinnvoll.

Eine wichtige Forderung an einen echtzeitfähigen Debugger oder an sonstige Diagnosemittel ist die: man muss sich in ein laufendes System einklinken können ohne die Anwendung neu starten zu müssen. Beim Einsatz des Diag-

nosemittels Logging empfiehlt es sich, das Logging über eine einfache Kommandoschnittstelle (z.B. einen UDP Listener) ein- oder ausschalten zu können. Beim Einsatz eines Debuggers muss sich dieser mit dem bereits laufenden Prozess verbinden können. Sehr oft werden hier Debugger verwendet, die z.B. eine serielle Schnittstelle als steuernden Kanal verwenden.

Natürlich gilt auch hier: je einfacher eine Anwendung konzipiert ist, um so einfacher gestaltet sich die Fehlersuche.

Im unten stehenden Beispiel wird für das Logging über UDP meine Klasse HU_TRACE verwendet, die portabel unter Linux und Windows eingesetzt werden kann.

Testfähigkeit

Eine Echtzeitanwendung muss von vornherein für Tests verschiedener Art – insbesondere für Lasttests und Tests von Grenzsituationen – vorbereitet werden. Dazu haben sich u.a. die folgenden Maßnahmen empfohlen:

- die Anwendung muss stabile Protokollierungsschnittstellen haben damit die Testergebnisse aufgezeichnet werden können. So ist z.B. die Protokollierung in Dateien keine gute Idee, wenn gestestet werden soll, wie sich das System bei vollem Dateisystem verhält.
- die Anwendung muss fernsteuerbar sein. So sollten z.B. alle Benutzereingaben über Tastatur auch über andere Schnittstellen wie z.B. UDP erlaubt sein. Damit können auch anormal hohe Eingabemengen und Eingabefrequenzen getestet werden. Über eine Fernsteuerschnittstelle ist auch automatisches Testen über Testscripts wesentlich einfacher.

Stabilität

Stabilität heißt: die Anwendung muss in jeder Situation – insbesondere in Fehlersituationen und in Lastsituationen stabil laufen. Dies kann nur dann einigermaßen verlässlich garantiert werden, wenn die Anwendung unter realistischen Bedingungen getestet wurde. Die Testfähigkeit ist also eine Voraussetzung für die Stabilität.

Bisweilen werden zur Erzielung von Stabilität Monitorprozesse eingesetzt, die eine gestorbene Anwendung automatisch wieder starten. Das ist ein schlechter Ansatz: er weist darauf hin, dass die Anwendung selbst instabil ist. Die Grundforderung muss immer sein: die Anwendung muss so stabil sein, dass sie durch übliche Fehlersituationen (z.B. unerwartete Ausnahmen, falsche Benutzereingaben) nicht abstürzt.

Für den Entwickler heißt das konkret: alle Fehlersituationen müssen ausprogrammiert werden – keine mögliche Fehlersituation darf unbehandelt

bleiben. Üblicherweise bedeutet dies:

- zur Entwurfszeit müssen alle Fehlerszenarien ins Design umgesetzt werden
- alles Ausnahmen („exceptions“) müssen abgefangen werden
- alle Return-Codes müssen analysiert werden
- ein klares und einheitliches Fehler-Logging muss von vorneherein geplant werden

Aufbau einer Echtzeitanwendung

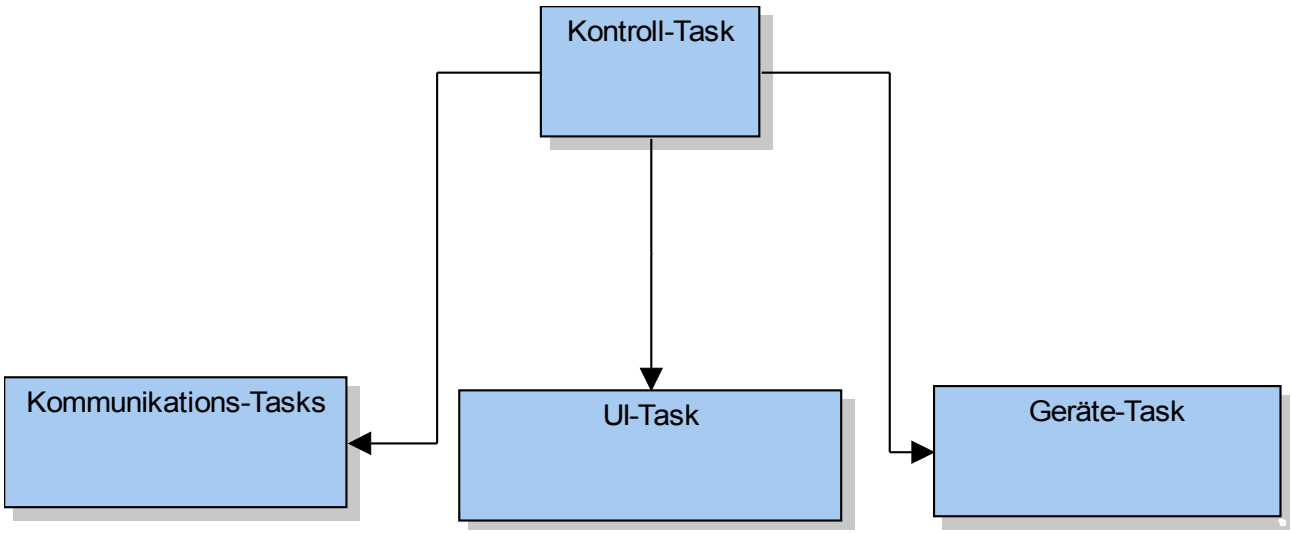
Da Echtzeitanwendungen bezgl. Testbarkeit, Verlässlichkeit, Reaktionszeit etc. anderen Anforderungen und andere Eigenschaften aufweisen als normale kommerzielle Anwendungen, ist eine klare Strukturierung besonders wichtig.

Eine Echtzeitanwendung besteht im allgemeinen aus mehreren Tasks, die oft als Threads realisiert werden. Sie können aber auch als Prozesse realisiert werden. Tasks sind vollgewichtige oder leichtgewichtige („Threads“) Systemprozesse, die sich um inhaltlich abgegrenzte Aufgaben kümmern. Im nachfolgenden Text gehen wir davon aus, dass die Tasks als Threads implementiert werden.

Echtzeitanwendungen bestehen meist aus mehreren Tasks, die praktisch immer paralle d.h. gleichzeitg ausgeführt werden. Die Aufteilung in Tasks ist meist inhaltlich motiviert. Eine typische Aufteilung kann z.B. so aussehen:

- Die **Haupttask** (auch Kontrolltask) genannt bildet die oberste Kontrollebene: sie startet und kontrolliert die darunter liegenden Tasks. Oft ist die Kontroll-Task nichts anderes als der Threads des Hauptprogramms (*main()*) des Gesamtprozesses.
- **Kommunikationstasks** versorgen die Gesamtanwendung mit Daten von anderen Systemen oder schaffen diese Daten zu anderen Systemen. Kommunikationstasks können vollständig fehlen, z.B. bei reinen Maschinensteuerungen.
- **Hardwaretasks** (Geräte-Tasks) reden mit den angeschlossenen Geräten
- **UI-Tasks** sorgen dafür, dass Informationen an den Benutzer ausgegeben werden. UI-Tasks können vollständig fehlen, wenn die Anwendungen als Services (Hintergrundprozesse) oder als einfache Konsolprozesse laufen.

Eine typische Task-Struktur kann also so aussehen:



Es nicht üblich, dass untergeordnete Tasks selbst weiter Unter-Tasks starten, also selbst zur nachgeordneten Kontroll-Task werden: dies würde dem Prinzip der Einfachkeit entgegenlaufen und sollte deshalb vermieden werden. Eine optimale Task-Hierarchie ist also nur zweistufig: es gibt nur eine Kontroll-Task – alle anderen Tasks sind dieser Kontroll-Task nachgeordnet.

Tasks bestehen fast immer aus einer Endlosschleife, in der allerdings die Kontrolle durch Aufruf von Betriebssystemroutinen (z.B. *sleep())* regelmäßig abgegeben wird. Eine typische Task-Schleife hat also den folgenden Aufbau:

```
while (true)
{
  if (Liegt_etwas_an()
  {
    tue_etwas();
  }
  sleep(2); // 2 = wert in ms – nur als Beispiel
}   // end while (true)
```

In der Praxis sollte noch - wie im Beispielsprogramm unten – eine Abfrage hinzugefügt werden, ob die Task sich selbst beenden soll.

Seit einiger Zeit gibt es in Windows und Unix/Linux auch andere Mechanismen, die das hier gezeigt Grundmuster (Nachschauen ob etwas zu tun ist, etwas tun) variieren: das sogenannte asynchrone IO (in alten Betriebssystem wie Vax/VMS übrigens ein alter Hut!). Dabei werden die „tue_etwas()" Calls nur aufgesetzt. Sie kehren sofort zurück, falls nichts zu tun ist. Sie senden Signale, wenn später etwas zu tun ist.

Informationsversorgung der Tasks

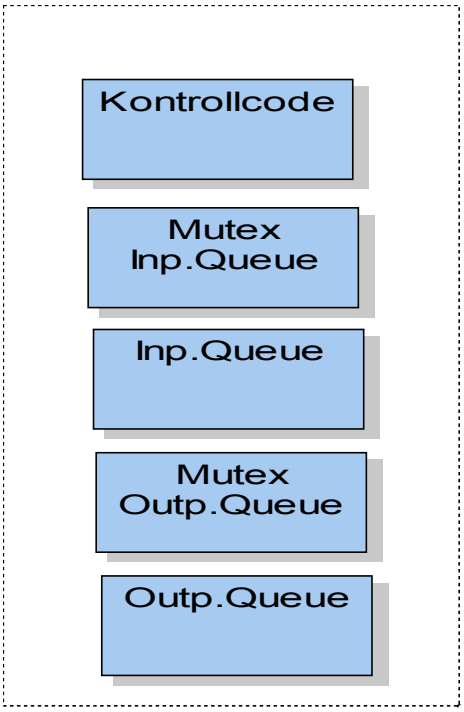
Als einfaches Mittel zur Informationsversorgung von Tasks haben sich **Queues** (Warteschlangen) bewährt. Eine Queue ist nichts anderes als eine einfach oder doppelt verkettete Liste von Informationen – meist als FIFO Stack oder auch als Datei realisiert. Die Informationen selbst kann man als Aufträge (weitere Begriffe dafür: Mitteilungen, Botschaften, Telegramme, Messages) oder für die jeweilige Task verstehen.

Die Queues werden doppelseitig benutzt: eine fremde Task (bisweilen auch die Eigner-Task) schreibt Aufträge oder Mitteilungen hinein, die Eigner-Task (un nur diese!) liest sie aus und führt die Aufträge aus. Da beide Vorgänge – Hin-einschreiben und Auslesen – potentiell gleichzeitig ablaufen können, müssen die Zugriffe auf die Queues gesperrt werden. Dies geschieht über einen **Mutex** (auch Semaphore sind nutzbar), der genau dieser Queue zugeordnet ist.

Die Verwendung von Queues führt zu zwei extrem wichtigen Eigenschaften:

- der seriellen (d.h. nicht-gleichzeitigen) Abarbeitung von Aufträgen. Seri-elle Abläufe sind bekanntlich besonders einfach zu verstehen: unser wichtigstes Entwurfskriterium – das der Einfachheit - wird also beachtet.
- der zweitweisen Speicherung von Aufträgen. Aufträge gehen nicht verloren, solange die Kapazität der Queue ausreicht.

Jede Task – auch eine übergeordnete Kontroll-Task - kann also als Kapsel aus mindestens fünf wesentlichen Teilen betrachtet werden:



Wenn eine Task einer anderen etwas mittzuteilen hat, dann kann dies auf zwei Arten geschehen:

- Task A stellt Task B eine Mitteilung in die Input-Queue von Task B
- Task A stellt eine Mitteilung für Task B in die Input-Queue der Kontroll-task (also in die Kontroll-Queue). Diese leitet die Meldung dann weiter an die Task B

Nehmen wir das erste Szenario. Task B hat als Input Queue die *QueueB*, der wiederum Mutex *mutex_qb* fest zugeordnet ist. Der Ablauf sieht dann in Pseudo-Code etwa so aus:

Task A:

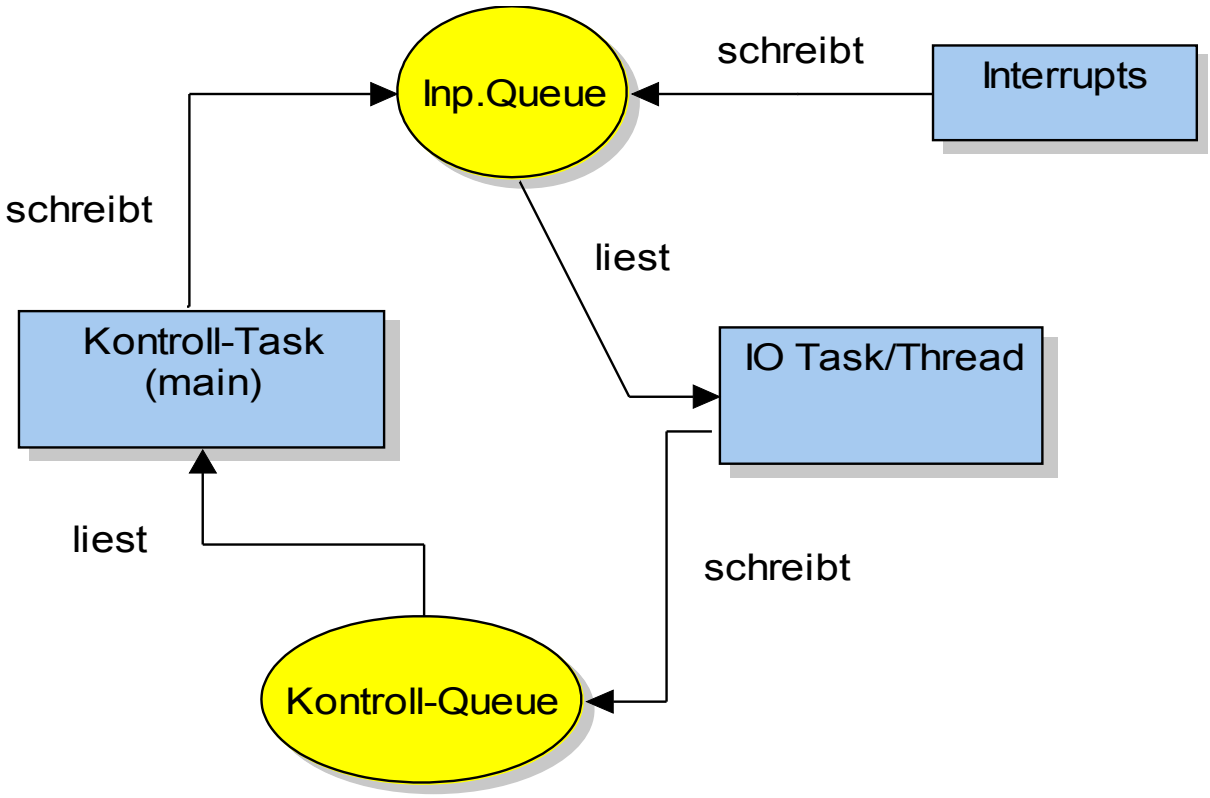
```
setlock(mutex_qb);
Schreiben Mitteilung in QueueB
unlock(mutex_qb);
```

Task B:

```
setlock(mutex_qb)
Lesen Mitteilung aus QueueB
unlock(mutex_qb)
```

Als besonders interessanten Fall der Kommunikation zwischen Tasks wird Beispielsprogramm unten die Abbruch-Mitteilung implementiert. Diese Mitteilung („exit“) wird an jede IO-Task geschickt, sobald ein Kontroll-C auf der Tastatur eraknnt wird.

Die typische Interaktion zwischen zwei Tasks ist im folgenden Diagramm am Beispiel der zentralen Kontroll-Task und einer IO-Task dargestellt. Im Diagramm wird übrigens eine **Interrupt-Serviceroutine** ähnlich wie ein Thread verwendet:



Auf jeden Fall sollte vermieden werden, dass Informationen auf andere Weise ausgetauscht werden, etwa durch Setzen globaler Variablen.

Für Queues und Mutexe gelten folgende Regeln:

- jeder Queue entspricht genau ein Mutex
- Jede Queue darf nur von einer Task gelesen (asugewertet) werden
- Durch Mutexe gesperrte Strecken müssen so kurz wie möglich sein
- Wenn Tasks der gleichen Ebene (z.B.IO-Tasks) miteinander kommunizieren müssen, sollte dies über die übergeordnete Kontrolltask geschehen.

Queues werden in Echtzeitsystemen insbesondere durch Interruptserviceroutinen oder durch Poll-Routinen gefüllt. Die Verwendung von Queues führt automatisch zur **Serialisierung** von Vorgängen innerhalb einer Task und damit zu einfach zu kontrollierenden Abläufen.

Starten und Beenden der Tasks

Alle Tasks dürfen nur aus der Kontroll-Task gestartet und auch nur von dort beendet werden – sonst ist der Begriff „Kontroll-Task" unsinnig. Die Tasks werden üblichweise zu zwei Zeitpunkten gestartet:

- zu Programmstart über einen einfachen Startaufruf im Hauprogramm der

Anwendung

- während des laufenden Betriebs, wenn bestimmte Bedingungen erfüllt sind. So muss z.B. eine Geräte Task z.B. für einem Motor erst dann gestartet werden, wenn der Motor angelaufen ist. Die Überwachung der Startbedingung für die Geräte – Task ist deshalb eine typische Aufghabe der Kontroll-Task

Ähnlich verhält es sich mit dem Beenden von Tasks. Auch hier sind drei Fälle unterscheidbar:

- alle Tasks werden bei Ende des Gesamtprogramms bzw. bei Ende der Kontroll-Task beendet. So kann z.B. das Drücken der Tastenkombination CTL-C das Ende des Hauptprogramms erfordern was wiederum zuvor das Beenden aller untergeordneten Tasks erfordert.

- die Kontroll-Task erkennt, dass eine untergeordnete Task beendet werden muss. Ein solcher Fall kann z.B. auftreten, wenn die regelmäßig von allen Tasks ausgesandten Kontrollmitteilungen („alive messages") für eine Task ausbleiben

- eine untergeordnete Task erkennt selbst, dass sie beendet werden muss. So kann z.B. die Task für eine Motorkontrolle erkennen, dass der Aus-Schalter betätigt wurde und dass es deshalb keinen Sinn mehr macht weiterzulaufen.

Es ist sehr wichtig, dass Tasks, die beendet werden, der Kontroll-Task mitteilen, wann sie definitiv „am Ende" sind. Normalerweise gibt es Abhängigkeiten inhaltlicher Art zwischen Tasks die es erfordern, dass Tasks in bestimmten Reihenfolgen gestartet und beendet werden.

In unserem Beispielsprogramm unten wird eine einfache Methode zum Beenden der Threads verwendet: jeder Thread wertet dazu ein **exit**-Kommando aus, das ihm in seiner Input-Queue übergeben wird. Bei Erkennen diese Kommandos beendet sich jeder Thread selbst. Da der Autrag zum Beenden von außen – durch die Kontroll-Task – erfolgt, erübrigt sich eine Mitteilung an die Kontrolltask.

Das Beenden von Threads ist leider etwas komplexer als das Starten. Details und alternative Lösungen können in Hart[2005] und Butenhof[1997] nachgeschaut werden.

Regeln für Prozeduren

Im Sinne des Hauptentwurfskriteriums gelten für Prozeduren (im OO Jargon „Methoden") besondere Regeln, die meist eingehalten werden können und fast

automatisch zu einem einfachen und sauberen Design führen:

- Prozeduren, die globale Ressourcen (Speicher, Geräte) verwenden, sollten nur von einer Task verwendet werden: damit sind Sperrmechanismen überflüssig.
- Bei gleichzeitiger Verwendung von Betriebsystemprozuren oder Bibliotheksprozeduren (z.B. printf() in C) muss darauf geachtet werden, dass die Prozeduren oder Bibliotheken threadfest sind.
- Wenn die gleichen Prozeduren trotzdem von mehreren Tasks aufgerufen werden sollen ist zu prüfen, ob nicht eine Verdoppelung der Prozedur die einfachere Lösung ist.

Eine beliebte aber unsaubere Lösung beim Pollen von Geräten stellen Prozeduren dar, die neben ihrer Hauptaufgabe – dem Abfragen eines Gerätestatus auch noch **Timeouts** überwachen. Typisches Beispiel: ein Bus wird gepollt und in der gleichen Routine wird auf die Antwort vom Bus gewartet. Diese Lösung vermischt zwei Dinge: die eigentliche Aufgabe und eine Überwachungsfunktion. Die große Gefahr liegt darin, dass bei der Timeout-Überwachung etwas schief läuft und dass die Routine dann nie mehr zurückkehrt. Außerdem muss die Kontrolle dann in der gerufenenen Prozedur abgegeben werden, was leicht zu undurchschaubaren Zeitverhältnissen führt. Solche Routinen können problemlos in zwei Routinen zerlegt werden:

- reine Abfrage des Gerätestatus
- Timoutüberwachung auf der Kontrollebene – wobei die Kontrollebene nicht unbedingt in der Kontrolltask angesiedelt sein muss.

In Pseudocode ergibt dies folgende Struktur in der Task Schleife:

```
while sub_task_go_on
{
    if (timeout_aktiv())
    {
      timeout_runtersetzen();
      if (timeout_abgelaufen()
         fehler_reaktion();
    }
    ret_code = polle_geraet();
    if (ret_code  < 0)
        timeout_starten();
    sleep(2);
}   // end while sub_task_go_on
```

Leider stelle ich in der Praxis immer wieder fest, dass in den Poll-Prozeduren Schleifen mit potentiellem Endloscharakter verwendet werden. Es wird darauf

vertraut, dass Geräte in endlicher Zeit einen Status liefern. In sicheren Echtzeitsystemen darf dies nicht sein! Jeder Poll-Vorgang sollte mit einer maximalen Wartezeit (Timeout) versehen sein, nach deren Ablauf die Poll-Prozedur einen geeigneten Return-Status zurückmeldet.

Die Thread-Routinen selbst müssen immer die Kontrolle in festen Zeitabständen abgeben. Der übliche Mechnismus hierfür die Verwendung von *sleep()* Aufrufen. Es gibt jedoch auch andere Lösungen wie z.B. das Senden von Signalen. Das Eintreffen von Signalen kann zu Prozess- oder Threadwechseln führen.

Sperrmechanismen für Codestrecken

Codestrecken – auch in Threads – können vom Scheduler des Betriebssystems unterbrochen werden, wenn er glaubt, die Zeit sei dafür reif. Einige Konstrukte führen fast immer zur Unterbrechung von Codestrecken:

- Aufruf von sleep()
- Aufruf von IO-Routinen
- Warten auf Signale

Nun gibt es Fälle, in denen eine Codestrecke keinesfalls unterbrochen werden darf – etwas um sehr schnell ablaufende Vorgänge zu steuern oder auch nur „mitzukriegen". Dafür gibt es „critical sections", die natürlich sehr sparsam eingesetzt werden müssen. In Windows sieht die Verwendung solcher Sperrmechanismen so aus:

```
static CRITICAL_SECTION crit_sect1;
EnterCriticalSection(&crit_sect1);
… zu schützender Code …
LeaveCriticalSection(&crit_sect1);
```

Der zu schützende Code sollte natürlicn keine Aufrufe von langlaufenden Routinen enthalten – sonst wird das Echtzeitverhalten (gefordert sind garantierte Antwortzeiten!) der Anwendung empfindlich gestört.

In Treibern findet man in Interruptserviceroutinen ein weiteres Konstrukt: das Verbieten und Wiedereinschalten von Interrrupts. Überlicherweise werden bei Betreten einer Interruptserviceroutine (diese wird normalerweise nicht im Code „aufgerufen", sondern bei Eintreffen eines Hardware-Ereignisses „betreten") die Interrupts ausgeschaltet und nach Abarbeiten einer möglichst kurzen Codestrecke wieder eingeschaltet. Der Sinn dieses Vorgehens liegt darin zu verhindern, dass während der Abarbeitung einer Interrupt-Routine die gleiche Routine nicht ein zweites Mal betreten wird. Überholvorgänge sollen also

verhindert werden. Falls in Interruptserviceroutinen zwischen dem Ausschalten und dem Wiedereinschalten der Interrupts ein Fehler auftritt, stirbt möglicherweise der ganze Prozess oder auch die ganze Maschinen – Interrupts werden ja nicht mehr zugelassen!

Zusammenfassung

Übergeordnetes Entwurfsziel jeder Echtzeitanwendung sollte die Einfachheit sein: nur dann sind die daraus ableitbaren Ziele wie Testfähigkeit, Debugfähigkeit und Stabilität erreichbar.

Jede Echtzzeitanwendung sollte aus einer Kontroll - Task und mehreren nachgeordneten Tasks bestehen. Die nachgeordneten Tasks sollten selbst keine weiteren Tasks kontrollieren

Jede Task besteht aus drei wesentlichen Blöcken:

Kontrollcode

Queue

Mutex für diese Queue

Der Einsatz spezieller Frameworks hat sich durch meine Erfahrungen nicht bewährt und ist nicht nötig. Eine kleine Menge portabler Klassen reicht für den Einsatz in Echtzeitsystemen aus. Frameworks tendieren dazu, zusätzliche Komplexität in die Anwendungen einzuschleusen und führen unnötige Abhängigkeiten von Produkten oder Herstellern ein. Alle gängigen Betriebssysteme bieten die wichtigsten Mechanismen zu Steuerung von Echteitanwendungen, nämlich Threads oder Prozesse, Mutexe, Warteschlangen. Die erwähnte kleine Menge portabler Klassen setzt auf diesen Betriebssystemmechanismen auf.

Ein Beispielsprogramm

Das folgende Bedispielsprogramm setzt die oben disktierten Prinzipien um. Es demonstriert, wie zwei Inputquellen – hier Tastatur und Kommandos über UDP (Netzwerk) in einer gemeinsamen Kontroll-Task ausgewertet werden. Drei Tasks werden als Threads realisiert:

Kontroll-Task: sie ist identisch mit dem Hauprogramm main(). Diese Task muss nicht explizit als Thread gestartet werden, weil jedes main()-Programm automatisch einen Thread startet. Die Kontrolltask wertet eine Queue aus, in der alle Programme landen, die zentral ausgewertet werden sollen.

Tastatur-Task: realisiert über Klasse KbThread. Sie nimmt Input von der Tastatur entgegen (Ausnahme: Abbruch über CTL-C) und leitet diesen Input zur Auswrtung an die Kontroll-Task weiter.

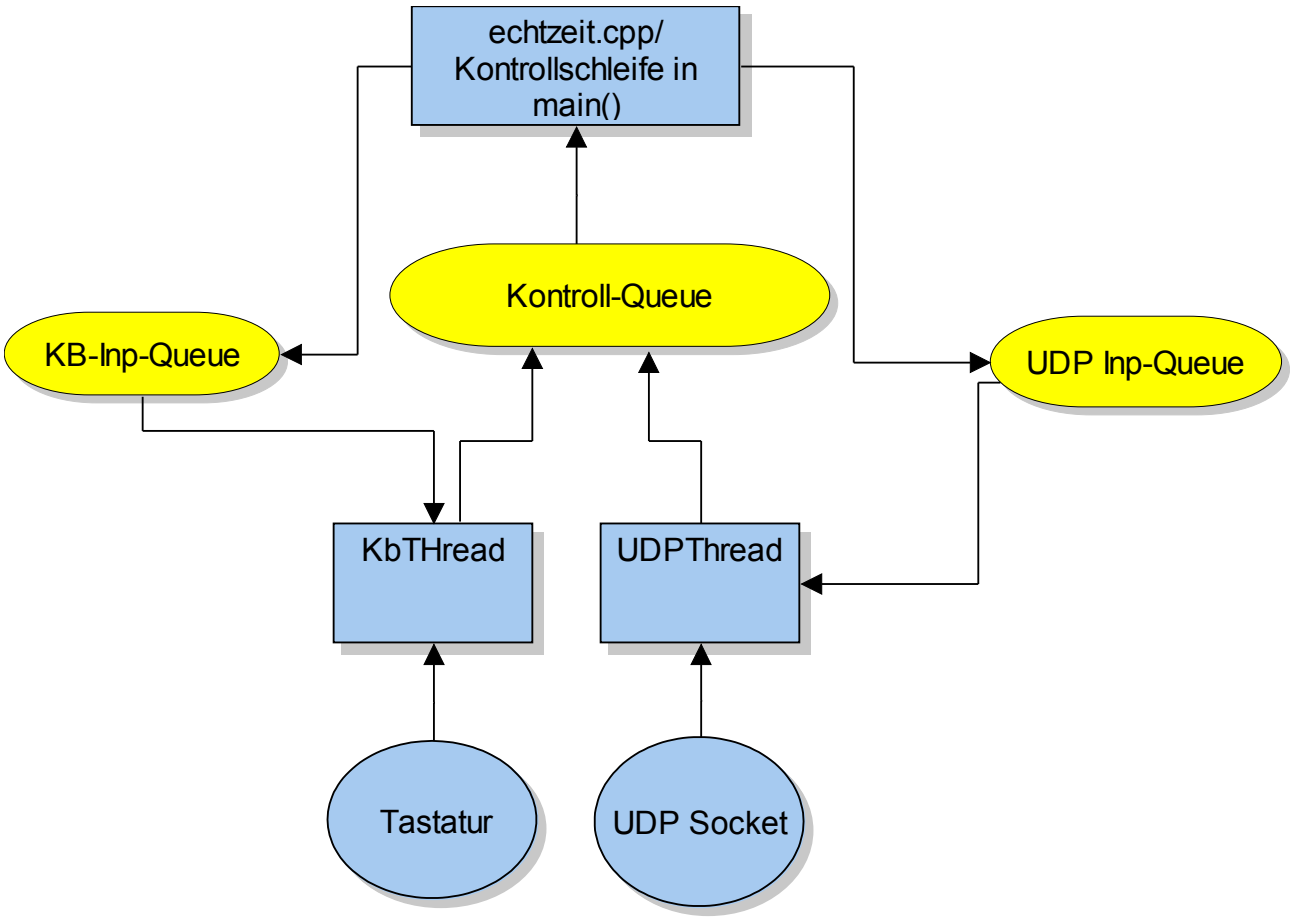
UDP-Task: realisiert über Klasse UDPThread. Sie nimmt Input über einen UDP Socket (Netzwerk) entgegen und leitet diesen Input zur Auswrtung an die Kontroll-Task weiter.

Warum werden die von Tastatur oder UDP stammenden Kommandos in die Kontroll-Queue weitergeleitet? Warum werden sie nicht sofort in den entsprechenden Threads ausgewertet? Ganz einfach: weil so eine einfache Serialisierung (im Sinne eines Nacheinander-Ausführens) erreicht wird. Diese Serialisierung ist sehr sinnvoll, falls potentiell auf die gleichen Ressourcen zugegriffen wird. So könnte z.B. sowohl über Tastatur und über UDP gleichartige Kommandos abegsetzt werden, die längere Laufzeiten haben und zusätzlich auf gemeinsame Ressourcen wie Dateien, Flash-Memories etc. zugreifen. Außerdem wird so einfach ischergestellt, dass die gleichen Kommandos über verschiedene Medien eingegeben werden können.

Die Kommandos werden normalerweise aus den Thread-Klassen (KbTHread und UDPThread) an die Kontroll-Task zur Auswertung weitergegeben. Eine Ausnahme bildet allerdings die Abbruch-Behandlung (CTL-C über Tastatur): hier wird die Kontrolle in Formt von **exit**-Kommandos an die Thread-Klassen weitergerereicht. Diese beenden sich selbst sobald sie ein exit-Kommando in ihrer Input-Queue erkennen. Dadurch entfallen sogenannte Join-Mechanismen für Threads, die von Linux und Windows extra bereitgestellt werden, um Threads zentral und sauber zu beenden.

Das Programm kann leicht erweitert werden um zusätzliche Tasks/Threads. Die Klassen für diese Tasks sollten aus der Basisklasse ThreadBase abgeleitet werden, da dadurch automatisch die nötigen Kommunikationsmechanismen für die Tasks (Queues und Mutexe) geerbt werden. Obwohl das Programm für Windows konzipiert ist, kann es ohne viel Aufwand nach Unix/Linux portiert werden.

Das folgende Diagramm zeigt das Zusammenspiel der wichtigsten Threads, Module und Klassen:



Bleibt noch die Frage offen, wie Interrupt-Serviveroutinen an die Tasks/Threads abgebunden werden können. Ganz einfach: jede Interruptserviceroutine (es sollte nur eine pro Thread geben) füllt die jeweilige Input-Queue des Threads.

Das Beispielsprogramm ist hier nicht komplett abgedruckt: weitere Hilfsklassen (z.B. **HU_Q** und **Mutex,** Logging) und Header befinden sich im begefügten ZIP-Archiv!

Die per default eingeschalteten **Loggingausgaben** können mit dem im zip-Archiv befindlichen Programm **udp_server** angeschaut und über Redirection ggf. in eine Datei umgeleitet werden.

Für die Übergabe von Kommandos per UDP kann das ebenfalls im ZIP-Archiv befindliche Programm **udp_client** verwendet werden. Der UDP Port ist auf 17777 voreingestellt.

Das Programm wurde mit dem C++ Compiler von Digital Mars (s.

www.digitalmars.com) übersetzt. Es enthält keine besonderen compilerabhängigen Konstrukte und sollte deshalb auch mit anderen C++ Compilern unter Windows erzeugt werden können. Für produktionsreife Anwendungen

sollten unbedingt die oben diskutierten Maßnahmen zur Ausnahmebehandlung (s. Huckert[2006]) eingebaut werden.

```
// Module echtzeit.cpp
// Copyright Dr.E.Huckert 12-2010
// Demonstrates:
//   - Creation, termination and use of two IO parallel threads
//   - Use of lock mechanisms (mutexes)
//   - Use of message queues between threads

// Compile (Digital Mars C++ compiler):
//    dmc -mn -D_WINSOCKAPI_ echtzeit.cpp mutex.cpp hu_q.cpp
//         threadbase.cpp kbthread.cpp udpthread.cpp
//         hu_trace.cpp wsock32.lib

#include <stdio.h>
#include "kbthread.h"
#include "udpthread.h"
#include "hu_q.h"
#include "mutex.h"
#include "hu_trace.h"

// global variables
HU_TRACE *pTrace     = new HU_TRACE("REALTIME", "localhost", 9514);
Mutex *pFlagMutex    = new Mutex("FLAGS");
int main_go_on       = 1;
Mutex *pMainM        = new Mutex("MAINQ");  // mutex for the main queue
HU_Q *pMainQ         = new HU_Q(100);       // main queue
//
Mutex *pKbInpM       = new Mutex("KBINP");  // mutex for the keyboard inp. queue
HU_Q *pKbInpQ        = new HU_Q(10);        // main keyboard inp. queue
KbThread *pKbThread = NULL;
//
Mutex *pUdpInpM       = new Mutex("UDPINP");  // mutex for the keyboard inp.
queue
HU_Q *pUdpInpQ       = new HU_Q(10);        // UDP inp. queue
UDPThread *pUdpThread = NULL;
int udpPort           = 17777;
```

```
// ------------------------------------------------------------------------
// CTRL-C Handler - is invoked when CTL-C is pressed
// We set here the flag for the main thread to 0 so that
// the main event loop is finished
BOOL WINAPI ctrlcHandler(DWORD event_type)
{
  if (::pTrace != NULL)
    ::pTrace->trace("ctrlcHandler() activated");
  ::pFlagMutex->setlock();
  ::main_go_on     = 0;
  ::pFlagMutex->unlock();
  return TRUE;
}   // end  ctrlcHandler()

// ------------------------------------------------------------
int evaluateCommandLine(int argc, char *argv[])
{
  int ret = 0;
  if (::pTrace != NULL)
    ::pTrace->trace("evaluateCommandLine()started");
  // ... nothing to do here at the moment
  return ret;
}   // end evaluateCommandLine()

// ------------------------------------------------------------
// Start all IO threads
// We start two threads here:
//   A keyboard input thread
//   A UDP input thread
// Note: A third thread - the main thread - is already started
int startIOThreads()
{
  int ret = 0;
  pKbThread = new KbThread(pKbInpQ, pKbInpM, pMainQ, pMainM);
  pKbThread->setLogging(::pTrace);
  pKbThread->createAndRun();
  if (::pTrace != NULL)
    ::pTrace->trace("startIOThreadS() thread 1 started");
  ret++;
  //
```

```
  pUdpThread = new UDPThread(::udpPort, pUdpInpQ, pUdpInpM, pMainQ, pMainM);
  pUdpThread->setLogging(::pTrace);
  pUdpThread->createAndRun();
  if (::pTrace != NULL)
    ::pTrace->trace("startIOThreadS() thread 2 started");
  ret++;
  return ret;
}   // end startIOThreads()

// ------------------------------------------------------------
// stop all IO threads
// We put "exit" commands into the input queue for the threads.
// These commands will be read by the threads - the thread will then
// stop themselves
int stopIOThreads()
{
  int ret = 0;
  ::pKbInpM->setlock();
  ::pKbInpQ->add("exit");
  ::pKbInpM->unlock();
  if (::pTrace != NULL)
    ::pTrace->trace("stopIOThreadS() thread 1 stopped");
  ret++;
  //
  ::pUdpInpM->setlock();
  ::pUdpInpQ->add("exit");
  ::pUdpInpM->unlock();
  if (::pTrace != NULL)
    ::pTrace->trace("stopIOThreadS() thread 2 stopped");
  ret++;
  return ret;
}   // end stopIOThreads()

// ------------------------------------------------------------
int main(int argc, char *argv[])
{
  int cRet;
  char *pCmd;//
  // evalute the command line
  cRet = evaluateCommandLine(argc, argv);
```

```
//
// start the IO threads
cRet = startIOThreads();
if (::pTrace != NULL)
  ::pTrace->trace("IO threads started result=", cRet);
//
// We are  here the main thread.
// We start the main event loop.
// Note: main_go_on is set to zero in the CTL-C handler
SetConsoleCtrlHandler(::ctrlcHandler, TRUE);
//
while (1)
{
  // go_on flag still set?
  ::pFlagMutex->setlock();
  if (! ::main_go_on)
  {
    ::pFlagMutex->unlock();
    break;
  }
  ::pFlagMutex->unlock();
  //
  // Inspect the main queue
  ::pMainM->setlock();
  cRet = pMainQ->getNoEntries();
  if (cRet > 0)
    // read the message in the main queue
    pCmd = pMainQ->get();
  else
    pCmd = NULL;
  ::pMainM->unlock();
  //
  if (pCmd != NULL)
  {
    // evaluate the message from the main queue
    if (::pTrace != NULL)
     ::pTrace->trace("main thread: message=", pCmd);
    delete pCmd;
  }
  //
```

```
    ::Sleep(100);
  }   // end while (1)
  //
  // stop all IO threads
  cRet = stopIOThreads();
  if (::pTrace != NULL)
    ::pTrace->trace("IO threads stopped result=", cRet);
  //
  // We sleep here in order to ensure that all threads are terminated
  ::Sleep(250);
  //
  if (pFlagMutex != NULL)
    delete pFlagMutex;
  if (pMainM != NULL)
    delete pMainM;
  if (pMainQ != NULL)
    delete pMainQ;
  if (::pTrace != NULL)
  {
    delete ::pTrace;
    ::pTrace = NULL;
  }
  ::exit(0);
}   // end main()


// module kbthread.cpp
// Copyright Dr.E.Huckert 12-2010
#include "kbthread.h"

// ----------------------------------------------------------
// Constructor
// Creates a thread instance - does not start the thread!
KbThread::KbThread(HU_Q *piq, Mutex *pim,
                   HU_Q *poq, Mutex *pom) :
                  ThreadBase(piq, pim, poq, pom)
{
  this->threadId = 0;
  this->pTrace = NULL;
}
```

```
// ------------------------------------------------------
void KbThread::setLogging(HU_TRACE *pt)
{
  this->pTrace = pt;
}

// ------------------------------------------------------
// thread worker routine
//void *threadWorker_kb(void *threadid)
DWORD WINAPI threadWorker_kb(void *param)
{
  int ret   = 0;
  int cRet;
  char msg[256];
  KbThread *pk;
  //
  // get a pointer the the KbThread instance
  pk = (KbThread *)param;
  //
  while (1)
  {
    ::Sleep(10);
    //
    // are there commands in out input queue?
    msg[0] = 0;
    (pk->getInpMutex())->setlock();
    cRet = (pk->getInpQueue())->getNoEntries();
    if (cRet > 0)
    {
      // read the message in the input queue
      char *pCmd = (pk->getInpQueue())->get();
      ::strcpy(msg, pCmd);
      delete pCmd;
    }
    (pk->getInpMutex())->unlock();
    if (::strlen(msg) > 0)
    {
      if (::strncmp(msg, "exit", 4) == 0)
        // "exit" command detected - leave this loop
```

```
        // leave this thread!
        break;
    }
    //
    // any events on the keyboard?
    if (kbhit())
    {
      fgets(msg, sizeof(msg), stdin);
      if (pk->pTrace != NULL)
        pk->pTrace->trace("threadWorker_kb(): msg=", msg);
      //
      // put the message just read into the queue for the main thread
      (pk->getOutMutex())->setlock();
      (pk->getOutQueue())->add(msg);
      (pk->getOutMutex())->unlock();
    }
  }   // end while (1)
  //
  if (pk->pTrace != NULL)
    pk->pTrace->trace("threadWorker_kb(): exiting");
  return (DWORD)ret;
}   // end threadWorker_kb()

// ------------------------------------------------------------
// create and start a thread
// Returns 0 if OK
int KbThread::createAndRun()
{
  int ret = 0;
  HANDLE hThread;
  // Start a thread that will read the DB queue and
  // execute the DB requests
  hThread = CreateThread(NULL,
                         (DWORD)(16L * 1024L),   // stack size in bytes
                         threadWorker_kb,        // thread worker routine
                         this,                   // argument to be passed
                         0,
                         &(this->threadId));
  if (this->pTrace != NULL)
      this->pTrace->trace("KbThread::createAndRun() ID=", (long)threadId);
```

```
  return ret;
}   // end KbThread::create()

// module UDPThread.cpp

#include "udpthread.h"

static WSADATA WSAdata;

// ----------------------------------------------------------
// Constructor
// Creates a thread instance - does not start the thread!
// Opens and binds a Datagramm (UDP) socket
// Note: if errors occur method getNoErrors() returns >0!
UDPThread::UDPThread(int udpPort,
                     HU_Q *piq, Mutex *pim,
                     HU_Q *poq, Mutex *pom) :
                     ThreadBase(piq, pim, poq, pom)
{
  this->threadId = 0;
  this->pTrace = NULL;
  this->errors = 0;
  this->port = udpPort;
  const int y = 1;
  // Windows needs this (no equivalent in Unix/Linux)
  if (WSAStartup(0x0101, &(::WSAdata)) != 0)
  {
    this->errors++;
    return;
  }
  // create and bind the socket
  this->sock = ::socket(AF_INET, SOCK_DGRAM, 0);
  if (this->sock < 0)
  {
    this->errors++;
    return;
  }
  ::setsockopt(this->sock, SOL_SOCKET,
              SO_REUSEADDR, (char *)(&y), sizeof(int));
  // Note: we bind here only for receiving - the send side would be
```

```
  // a little bit more complex
  ::memset(&(this->udpSender), 0, sizeof(this->udpSender));
  this->udpSender.sin_family      = AF_INET;
  this->udpSender.sin_port        = ::htons(this->port);
  if (::bind(this->sock,
            (struct sockaddr*)&(this->udpSender), sizeof(this->udpSender)) < 0)
    this->errors++;
}

// ------------------------------------------------------
UDPThread::~UDPThread()
{
  ::closesocket(this->sock);
}

// ------------------------------------------------------
void UDPThread::setLogging(HU_TRACE *pt)
{
  this->pTrace = pt;
}

// ------------------------------------------------------------------
// Receive from the socket
// Returns: < 0 upon error
//          = 0: nothing received
//          > 0: something received
int UDPThread::receive(char *data, size_t size)
{
   int len, ret;
   ::memset(data, 0, size);
   len = sizeof (this->udpSender);
   ret = ::recvfrom(this->sock, data, size, 0,
                    (struct sockaddr *)&(this->udpSender), &len);
   if (ret == SOCKET_ERROR)
     ret = -1;
   else
     ret = len;
   return ret;
}   // end UDPThread::receive()
```

```
// ------------------------------------------------
// thread worker routine
//void *threadWorker_udp(void *threadid)
DWORD WINAPI threadWorker_udp(void *param)
{
  int ret   = 0;
  int cRet;
  char msg[256];
  UDPThread *pk;
  //
  // get a pointer the the UDPThread instance
  pk = (UDPThread *)param;
  //
  while (1)
  {
    ::Sleep(10);
    //
    // are there commands in out input queue?
    msg[0] = 0;
    (pk->getInpMutex())->setlock();
    cRet = (pk->getInpQueue())->getNoEntries();
    if (cRet > 0)
    {
      // read the message in the input queue
      char *pCmd = (pk->getInpQueue())->get();
      ::strcpy(msg, pCmd);
      delete pCmd;
    }
    (pk->getInpMutex())->unlock();
    if (::strlen(msg) > 0)
    {
      if (::strncmp(msg, "exit", 4) == 0)
        // "exit" command detected - leave this loop
        // leave this thread!
        break;
    }
    //
    // receive via UDP
    cRet = pk->receive(msg, sizeof(msg));
    if (cRet > 0)
```

```
    {
      if (pk->pTrace != NULL)
        pk->pTrace->trace("threadWorker_udp(): RX=", msg);
      //
      // put the message just read into the queue for the main thread
      (pk->getOutMutex())->setlock();
      (pk->getOutQueue())->add(msg);
      (pk->getOutMutex())->unlock();
    }
  }   // end while (1)
  //
  if (pk->pTrace != NULL)
    pk->pTrace->trace("threadWorker_udp(): exiting");
  return (DWORD)ret;
}   // end threadWorker_udp()

// ------------------------------------------------------------
// create and start a thread
// Returns 0 if OK
int UDPThread::createAndRun()
{
  int ret = 0;
  HANDLE hThread;
  // Start a thread that will read the DB queue and
  // execute the DB requests
  hThread = CreateThread(NULL,
                         (DWORD)(16L * 1024L),   // stack size in bytes
                         threadWorker_udp,        // thread worker routine
                         this,                   // argument to be passed
                         0,
                         &(this->threadId));
  if (this->pTrace != NULL)
      this->pTrace->trace("UDPThread::createAndRun() ID=", (long)threadId);
  return ret;
}   // end UDPThread::create()
```

Literatur

Butenhof, David R.: Programming with POSIX Threads, Boston: Addison Wesley, 1997

Bauke, Heiko/**Mertens,** Stephan: Cluster Computing, Berlin, Heidelberg: Spirnger, 2006

Hart, Johnson M.: Windows System Programming, Boston: Addison Wesley, 2005

Huckert, Edgar: Ausnahmebehandlung in C++ , Windows und Linux, 2006, s. http://www.huckert.com/ehuckert/exceptions.rtf