Joachim Wietzke

Ergänzungen zu: Automotive Embedded Systeme, Effizientes Framework - Vom Design zur Implementierung, Edition xx

4. September 2005

Springer
Berlin Heidelberg New York
Hong Kong London
Milan Paris Tokyo

Inhaltsverzeichnis

Die Listings

Abbildungsverzeichnis

1

kleine Tips und Tricks, die im Buch keinen Platz hatten

1.1 Lazy Initialisation, Load on Demand

Werden in Projekten alle Objekte erst beim Start oder zur Laufzeit angelegt, so kommt es leicht zu inakzeptablen Startzeiten. Der übliche Trick ist dann, Objekte erst dann dynamisch anzulegen, wenn sie gebraucht werden. Man arbeitet in der Init-Phase auf dem Daten-Segment und in der Main auf dem Stack oder dem Heap.

Man nennt dies „lazy Initialisation“ oder auch „Load/Init on Demand“. Ein solches Vorgehen kann folgende Nachteile haben:

- Es gibt einen Wettlauf zwischen Objekten oder Komponenten.
- Objekte werden angesprochen, die noch nicht existieren.
- Um ein Fehlerphänomen zu verstehen, muss die gesamte Vorgeschichte beobachtet und verstanden werden.
- Diese Aufstartzeit ist nicht mehr deterministisch, das Verhalten im vernetzten System wird schwer kontrollierbar.
- Es gibt Phänomene, bei denen das Gerät regelmäßig 10 Sekunden Startzeit aufweist, nur einmal am Tag braucht es 30 Sekunden, und keiner weiß wieso.

Fazit: Die bessere, bereits vorgestellte Lösung ist, Objekte statisch im Code- oder im Datensegment anzulegen. Die Startzeit muss und kann auch ohne Tricks akzeptabel sein.

1.2 Nullpointer kontrollieren

Mit der bereits vorgestellten Überladung des new-Operators kann man auch überprüfen oder sicherstellen, dass nach jedem new-Aufruf der erzeugte Pointer auch zunächst auf NULL gesetzt wird. Entwickler vergessen das gerne und erzeugen damit schwer zu findende vagabundierende Pointer.

1.3 Virtual Destructor

Der Destruktor wird immer dann virtual definiert, wenn vererbt wird und wenn gilt:

- Objekte der abgeleiteten Klassen werden über Referenzen bzw. Zeiger der Basisklasse angesprochen (zum Beispiel ein Container für CEvent).
- Diese Objekte werden zerstört.
- Objekte der abgeleiteten Klassen haben einen Zeiger, für den ein delete-Aufruf zwingend ist.

Das Problem ist, dass der Autor einer Basisklasse nicht absehen kann, ob einmal geerbt wird und Objekte der abgeleiteten Klassen wie oben beschrieben angesprochen werden. Ein virtueller Destruktor ist deshalb bei einer einfachen Vererbung immer notwendig, da sonst eventuell unbeabsichtigt Objekte nicht freigegeben werden. Wenn zum Beispiel Klasse B von Klasse A erbt und Klasse C von Klasse B, würden B- oder C-Objekte, die über Pointer oder Referenzen vom Typ der Basisklasse freigegeben werden, nur A-Destruktoren erreichen.

Schlimmer noch führt dieser Versuch, ein abgeleitetes Objekt durch Aufruf des Basisklassendenstruktors zu löschen, zu einer undefinierten Situation [**?**].

Wollen wir die Vererbung ausdrücklich verbieten, so müssen wir die Konstruktoren als privat deklarieren und blockieren und die Objekte über eine Factory-Methode erzeugen [**?**].

```
class CNoinheritance

public:

CNoInteritance(const CNoInheritance&) // eigener copy ctor oder weglassen

static CNoinheritance getObject();       // eine Art Factory

.....

privat:

CNoInheritance (){....}}; //ctor blockieren

CNoInheritance& operator=(const CNoInheritance& rhs); //zuw blockieren
```

Listing 1-1. Blockieren der Konstruktoren

1.4 Kleine Auffrischung der static Deklaration

Static Konstanten sind nur innerhalb der eigenen Datei bekannt, modulglobal.

```
static const char* TUNER_COMPONENT_NAME = "TunerComponent";
```

Listing 1-2. statische Konstante

Man nutzt dies, um Namenskonflikte in großen Projekten zu vermeiden.

Lokale static Variable sind dauerhafte Variable und sind innerhalb ihrer eigenen Funktion bekannt. Globale static Variable sind dauerhafte Variablen und sind innerhalb ihrer Datei bekannt (modulglobal).

Static Variable behalten ihre Werte zwischen den Aufrufen bei. Die Deklaration und Definition einer static Variablen wird nur beim ersten Aufruf ausgeführt.

```
int myFunc
{
static int Number = 55; // nur beim ersten Aufruf
Number = Number +2;
return (Number);
}
```

Listing 1-3. statische Variable

Innerhalb von Klassendefinitionen können Variable oder Methoden static deklariert werden.

Eine static Elementvariable gibt es innerhalb ihrer Datei nur einmal, auch wenn mehrere Objekte instantiiert werden. Man nennt sie Klassenvariable. Sie ist modulglobal.

Alle Objekte verwenden die gleiche Variable, die mit Null initialisiert wird, bevor das erste Objekt erzeugt wird. Durch die Deklaration innerhalb der Klasse wird noch kein Speicher allokiert, erst durch Bereitstellung einer Definition außerhalb der Klasse wird Speicher im Datensegment bereitgestellt.

Klassenvariable werden mit dem Klassennamen und Bereichsauflösungs-Operator aufgerufen oder in Verbindung mit einem Objekt.

```
Klassenname::Klassenvariable
```

Listing 1-4. statische Klassenvariable

Auch Elementfunktionen können static deklariert werden. Sie werden damit zu Klassenfunktionen. Eine Klassenfunktion hat keinen this-Zeiger. Sie kann mit

```
Klassenname::Klassenfunktion
```

Listing 1-5. statische Klassenmethode

aufgerufen werden.

Schließlich können ganze Objekte static instantiiert werden:

```
static datentyp name
```

Listing 1-6. statische Objekte

Dann ist das ganze Objekt einschließlich der Methoden und Variablen modulglobal.

Eine Methode kann dann mit

```
Objekt.Methode
```

Listing 1-7. Aufruf eines static Objekts

aufgerufen werden.

Wird ein static Objekt innerhalb einer *.h Datei instantiiert, entstehen durch die Inkludierung in mehreren Modulen auch mehrere Objekte. So ist es meistens nicht gedacht.

Wird ein statisches Objekt instatiiert und danach der Prozess geforkt, so entstehen ebenfalls mehrere Objekte gleichen Namens. Ob das so gemeint war, ist zu prüfen.

Je nachdem, wie static Objekte deklariert und definiert wurden, können sie global in einem oder in allen Threads des Programms sein.

Eine andere Möglichkeit zur Erzeugung eines globalen Objekts ist die Deklaration in der *.h als

```
extern datentyp name
```

Listing 1-8. globales Objekt durch extern Deklaration

In einer *.cpp folgt die Instaniierung als

```
datentyp name
```

Listing 1-9. Instantiierung des globalen Objekts

Damit gibt es ebenfalls nur ein Objekt, das allen, die das *.h-File inkludieren, bekannt ist.

Diese Version ist etwas weniger aufwändig als die static-Version, da dort bei jedem Aufruf geprüft werden muss, ob es der erste Aufruf ist.

1.5 include guard

Include-Guards verhindern über Direktiven das mehrfache Inkludieren von Header-Dateien. Beim ersten Inkludieren einer Datei wird eine namensähnliche Variable definiert, deren Vorhandensein jedes weitere Inkludieren verhindert. Der Mechanismus kann vor dem Deploy der Datei überprüft werden oder in der inkludierten Datei selbst. Bei größeren Systemen kann die zweite Version zur spürbaren Verlängerung der Kompilierzeiten führen.

Vor der Inkludierung:

```
#ifndef _INKLUDEFILE_H
#define _INKLUDEFILE_H
#include "inkludefile.h"
#endif
```

Listing 1-10. include guard vor der Inkludierung

In der inkludierten Datei:

```
#ifndef _INKLUDEFILE_H
#define _INKLUDEFILE_H

.....
#endif
```

Listing 1-11. include guard in der Datei

1.6 class oder struct

Struct ist eine Klasse mit Elementen, die ausschließlich public sind. Dies betrifft MemberVariable und Methoden. Struct erlaubt keine Kapselung. Deshalb sollten Structs nur in Basisklassen verwendet werden.

1.7 union oder reinterpret_cast

Mithilfe einer `union` kann man dieselben Daten über mehrere Datenstrukturen ansprechen [**?**]. Ebenso gut kann man `reinterpret_cast` verwenden. Beide Konstrukte verhalten sich gleich und haben dieselbe Grundlage. Dementsprechend kann auch ein reinterpret_cast alignment-Probleme haben oder Probleme mit Zero-Padding.

1.8 Enum Index für Arrays

Statische Elemente können in einer Klassendefinition nicht definiert, sondern nur deklariert werden (Ausnahme integrale Typen bei neuen Compilern). Sie werden dann in der Implementierung in der Definition zugewiesen. Mit dem so genannten Enum-Hack können Konstanten schon in der Klassendefinition zugewiesen werden. [**?**]. Im unten stehenden Beispiel wird implizit von 0 an aufsteigend initialisiert.

Mit dem Enum Index kann man dann Arrays oder Tabellen namentlich adressieren.

```
enum
{
AUDIODISCPLAYER_INDEX,
AUDIODISCPLAYERDEVICE_INDEX,
KEYEVENTDISPATCHER_INDEX,
....

DEBUG_INDEX,

MOSTDISPATCHER_INDEX,
};
```

Listing 1-12. Enum-Hack

Mit diesem Namens-Index kann nun z.B. in nachfolgendem Array adressiert werden, siehe auch [**?**].

```
return sContextTable[AUDIODISCPLAYER_INDEX].mLocal;
```

Listing 1-13. Indexierung eines Arrays mit Enum-Konstanten

2

Zentrale Registrierung

Eine zentrale Registrierung der Komponenten ist an irgendeiner Stelle im System erforderlich, da die logischen Adressen (FunctionBlock, InstID) mit den physikalischen Knoten-Adressen im Netz verknüpft werden müssen. Viele der externen Devices haben dynamische Adressen, die vom Most-Master vergeben werden und von der Aufstart-Reihenfolge abhängen. Deshalb kann man die Adressen nicht statisch vergeben, auch wenn dies für Aufstartgeschwindigkeit und Systemsicherheit vernünftig wäre.

Es gibt in MOST-Systemen die so genannten NetServices, die neben Fehlerbehandlung und anderen Services auch die Adressvergabe übernehmen. Sie spiegeln die zentrale Registry, die sich im Most-Master befindet, in der lokalen Komponente und setzen die Adressen in MOST-Nachrichten um, bevor sie über den Treiber in den Ring gesendet werden.

In dem vorliegenden Framework wird der Einfachheit halber auf die NetServices verzichtet und der Treiber direkt vom Dispatcher angesprochen. Damit entfallen viele Sicherheitsmechanismen bei ungewollten Unlocks und eben auch die Umsetzung in den lokalen Registries.

Deshalb muss in der Headunit eine kleine Registry implementiert werden. Da dies für derartige Systeme untypisch ist, behandeln wir das Thema nur sehr komprimiert.

Die zentrale Registry-Komponente scannt zyklisch den MOST-Adressraum nach neuen Devices. Wird sie fündig, setzt sie die entsprechenden Adressen in den logischen Devices. Die Models merken sich die Adresse und verwenden sie für ihre MOST-Kommunikation. Wie oben beschrieben, werden normalerweise solche Aufgaben in den Protokoll-Schichten erledigt und nicht auf dem Applikationslevel.

Zunächst die Prototypen-Datei:

```
class CSoftTimer;
```

```
class CCentralregistryComponent: public AComponentBase,
                                 public ITimerListener,
                                 public ICentralregistryRequestPort,
                                 public ICentralregistryUpdatePort
{
   friend class CCentralregistryMostHandler;
public:
   CCentralregistryComponent(CComponentContext& theContext,
                   Int32 theTimerManagerStartCapacity);
   virtual ~CCentralregistryComponent();

   virtual void run();
   // prepare steps for possible shutdown
   virtual void prepareForShutdown();
   virtual void stop();
   virtual void init();            // initialize the component
   virtual void reRun();           // request for resume
   virtual void goToStandby();     // request go to standby mode
   // called by CSoftTimer
   virtual void processTimeOut(CSoftTimer& theTimer);
   virtual void addrUpdate();

private:
   void scan();
   void cdAddr(UInt32 theAddress);  // our external devices
   void tunerAddr(UInt32 theAddress);
   void naviAddr(UInt32 theAddress);
   void ampAddr(UInt32 theAddress);
   CSoftTimer * mTimerPtr;
   UInt32 mNaviAddr;
   UInt32 mCDAddr;
   UInt32 mTunerAddr;
   UInt32 mAmpAddr;
   static UInt32 sPollInterval;
   static UInt32 sAddrStart;
   static UInt32 sAddrEnd;
   bool mLockSent;
};
```

Listing 2-1. `CentralRegistryComponent.h`

Wir lassen hier dynamische Allokation ausnahmsweise zu, da sie ganz am Anfang des Programms geschieht und der Speicher nicht mehr freigegeben wird. Die Implementierung:

```
// parameters for fblockid scans:
```

```
UInt32 CCentralregistryComponent::sPollInterval = 10000; //milliseconds
UInt32 CCentralregistryComponent::sAddrStart = 0x100;
UInt32 CCentralregistryComponent::sAddrEnd = 0x100;
// used to generate application messages sent to MOST
static CMostEvent event;

CCentralregistryComponent::
CCentralregistryComponent(CComponentContext& theContext,
                          Int32 theInternalQueueSize,
                          Int32 theTimerManagerStartCapacity)
   : AComponentBase( theContext, theTimerManagerStartCapacity, 1),
     mNaviAddr(0), mCDAddr(0), mTunerAddr(0),
     mAmpAddr(0), mLockSent(false)
{
   mTimerPtr = NEW_STATIC(CSoftTimer, (this, sPollInterval, 0));
}

CCentralregistryComponent::~CCentralregistryComponent()
{
   DELETE_STATIC(mTimerPtr);
}

void CCentralregistryComponent::init(void)
{
   AComponentBase::init();
   CCentralregistryMostHandler * mostHandler =
      NEW_STATIC(CCentralregistryMostHandler, (*this));

   // register own listeners in local dispatcher
   CComponentServices::addListener
      (CMessage::REQUEST_TYPE,
       NEW_STATIC(CCentralregistryRequestHandler,(*this)));
   CComponentServices::addListener
      (CMessage::UPDATE_TYPE,
       NEW_STATIC(CCentralregistryUpdateHandler,(*this)));
   CComponentServices::addListener
      (CMessage::MOST_APPMSG_TYPE, mostHandler);
   CComponentServices::addListener
      (CMessage::MOST_POOL_REF_TYPE, mostHandler);
   scan();      // check for ext. devices
}

void CCentralregistryComponent::run(void)
{
   sOwnCout() << "Component Centralregistry has started" << endl;
   mTimerPtr->start();
   AComponentBase::run();
}
void CCentralregistryComponent::prepareForShutdown() {}
```

```
void CCentralregistryComponent::stop()
{
   mTimerPtr->stop();
}
void CCentralregistryComponent::reRun() {}
void CCentralregistryComponent::goToStandby(){}
//once a while
void CCentralregistryComponent::processTimeOut(CSoftTimer& theTimer)
{
   scan();
}
void CCentralregistryComponent::scan()
{
   for (UInt32 addr = sAddrStart; addr <= sAddrEnd; addr ++)
   {                                                  // scan devices
      event.setMostContent
         (addr,              // address of destination
          0x01,              // function block ID, Netblock
          0x00,              // instance ID
          0x000,             // function ID
          0x01,              // operation get FBlockIDs
          0x00               // length of following data
          );
      CContext::sendToMost (event.getMessage());
   }
}
void CCentralregistryComponent::cdAddr(UInt32 theAddress)
{
   if (mCDAddr != theAddress)
   {
      char buffer[] = "0x00000000";
      CAudiodiscplayerRequestPort::getInstance().
         setAddress(theAddress);
      mCDAddr = theAddress;
      sprintf(buffer, "%x", theAddress);
      sOwnCout() << "CD: MOST address " << buffer << endl;
   }
   addrUpdate();
}
void CCentralregistryComponent::tunerAddr(UInt32 theAddress)
{
   if (mTunerAddr != theAddress)
   {
      char buffer[] = "0x00000000";
      CAmfmtunerRequestPort::getInstance().
         setAddress(theAddress);
      mTunerAddr = theAddress;
      sprintf(buffer, "%x", theAddress);
      sOwnCout() << "Tuner: MOST address " << buffer << endl;
```

```
   }
   addrUpdate();
}
void CCentralregistryComponent::naviAddr(UInt32 theAddress)
{
   if (mNaviAddr != theAddress)
   {
      char buffer[] = "0x00000000";
      CNavRequestPort::getInstance().
         setAddress(theAddress);
      mNaviAddr = theAddress;
      sprintf(buffer, "%x", theAddress);
      sOwnCout() << "Navi: MOST address " << buffer << endl;
   }
   addrUpdate();
}

void CCentralregistryComponent::ampAddr(UInt32 theAddress)
{
   if (mAmpAddr != theAddress)
   {
      char buffer[] = "0x00000000";
      CAudioamplifierRequestPort::getInstance().
         setAddress(theAddress);
      mAmpAddr = theAddress;
      sprintf(buffer, "%x", theAddress);
      sOwnCout() << "Amp: MOST address " << buffer << endl;
   }
   addrUpdate();
}

void CCentralregistryComponent::addrUpdate()
{
   if (!mLockSent && mAmpAddr != 0 && mTunerAddr != 0)
   {
      CMessage msg(CMessage::MOST_LOCK_TYPE);
      CContext::getMostdispatcherContext().getSystemQueue().add(msg);
      mLockSent = true;
   }
}
```

Listing 2-2. `CentralRegistryComponent.cpp`

Vom Komponenten-Dispatcher werden die externen MOST-Nachrichten an den registrierten MOST-Händler weitergegeben.

```
class CCentralregistryComponent;
```

```
class CCentralregistryMostHandler : public IMessageHandler
{
public:
   CCentralregistryMostHandler(CCentralregistryComponent& component);
   void handleMessage(const CMessage& msg);
   virtual ~CCentralregistryMostHandler();
private:
   CCentralregistryComponent& mCentralregistry;
};
```

Listing 2-3. `CCentralRegistryMostHandler.h`

In der Implementierung wird auf FktID=0 und OpType=0c gesucht. Dies ist die Header-Information einer Statusrückmeldung eines physikalischen Geräts, in der alle Funktionsblöcke, die das Gerät bedient, nacheinander aufgelistet werden. Uns interessieren in dem vorliegenden kleinen Framework nur der externe Tuner, die externe Navigation und der externe Verstärker. Deshalb wird auch nur auf diese FBlöckIDs überprüft und bei Übereinstimmung die Geräteadresse (DeviceID) des externen Geräts gespeichert. Im MOST-Codec des logDevices wird diese Information zum Zusammenbau einer MOST-Nachricht an dieses Gerät verwendet. Nochmal sei betont, dass normalerweise Applikationen nicht physikalisch adressieren sollten, die Umsetzung aus der logischen Adresse wird in größeren Systemen in den unteren Protokollschichten vorgenommen.

```
CCentralregistryMostHandler::
CCentralregistryMostHandler(CCentralregistryComponent& component)
   : mCentralregistry(component)
{
}
CCentralregistryMostHandler::~CCentralregistryMostHandler()
{
}

void CCentralregistryMostHandler::handleMessage(const CMessage& aMsg)
{
  pMostMsgUnion msg = (pMostMsgUnion)aMsg.getMostMsgPtr();
  UInt32 funcId   = mostappmsg_getFuncID(msg);
  UInt8  opType   = mostappmsg_getOp(msg);

  if (msg->type == CMessage::MOST_APPMSG_TYPE)
  {
     // status of external device
     if (  0x000 == funcId && 0x0c == opType )
```

```
    {
       //example data (offset+7): 4001 0200 0401 0301 2201 0801
       UInt8 datalen = msg->length - 7;
       for(int i = 0; i < datalen; i+=2) // jump over instanceIDs
       {
         if (msg->u.am.Data[i] == CContext::getAmfmtunerFBlockID())
         {
            mCentralregistry.tunerAddr
               (ENDIAN_BE16(msg->u.am.Device_ID));
         }
         if(msg->u.am.Data[i] == CContext::getNavFBlockID())
         {
            mCentralregistry.naviAddr
               (ENDIAN_BE16(msg->u.am.Device_ID));
         }
         if(msg->u.am.Data[i]==CContext::getAudiodiscplayerFBlockID())
         {
            mCentralregistry.cdAddr
               (ENDIAN_BE16(msg->u.am.Device_ID));
         }
         if(msg->u.am.Data[i]== CContext::getAudioamplifierFBlockID())
         {
            mCentralregistry.ampAddr
               (ENDIAN_BE16(msg->u.am.Device_ID));
         }
       }
    }
  }
}
```

Listing 2-4. `CCentralRegistryMostHandler.cpp`

3

all inclusive

Wir haben in allen Quellen, um Wiederholungen zu vermeiden und Platz zu sparen, sämtliche #includes weggelassen. An dieser Stelle listen wir alle includes auf, die wir innerhalb des Frameworks brauchen, mit einem Kommentar, wofür. Hier folgt eine Übersicht der im Framework verwendeten includes, die mit dem üblichen ifdef komplett eingefügt werden könnten.

```
// Fehlerbehandlungen
#include <assert.h>                 // Assertions
#include <errno.h>                  // for the error constants

// System
#include <iostream>                 // standard io
#inlude   <stdio.h>                 // standard io
#include <stdlib.h>                 // for malloc in debug and virtual component
                                    // QNX also for mediaplayer
#include <signal.h>                 // ctrl_c and others and kill

#include <math.h>                   // standard math
#include <sys/time.h>               // because of time used

// Memory
#include <pthread.h>                // semaphore must be simulated, Posix Thread-Header
#include <new>                      // for placement new used in CObjectPool
#include <string.h>                 // for memset, memcpy
#include <stddef.h>                 // for size_t  eg. in CObjectPool

#include <vector>                   // needed in PosixShm
#include <sys/types.h>              // defines types used in defining values returned by system
                                    // level calls for file status and time information [System V]
#include <sys/stat.h>               // defines the structure used by the _stat() and _fstat()
```

```
                                    // routines  [System V]
                                    // specials
#include <stdarg.h>                 // defines ANSI-style macros for accessing arguments
                                    // of functions which take a variable number of arguments.

// QNX special for CD
#include "sys/cdrom.h"
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/dcmd_cam.h>
#define CD_DEVICE "/dev/cd0"        // for example

// QNX special for RS232
#include <pthread.h>
#include <sys/ioctl.h>              // serielle
#include <termios.h>                // serielle
#include <fcntl.h>                  // serielle

// GNU special for socket
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <unistd.h>                 // GNU
#include <fcntl.h>                  // GNU

// WIN32 socket
#include <winsock2.h>
#include <windows.h>                // for Thread Definitions
#include <sys/types.h>
#include <io.h>                     // declarations for low-level file handling + I/O fnctns
#include <time.h>

//      own specials for MOST
#include "mostmsg.h"
#include "mostnewif.h"
#include "mostutils.h"
#include "mostpool.h"               // for mostpool_open()
```

Listing 3-1. `CCentralRegistryMostHandler.h`

Literatur

[Bus] Buschmann, F.: Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M.
Pattern-Oriented Software Architecture; 1996; Wiley

[Doug] Douglass, Bruce Powel:
Real-Time Design Patterns; 2003; Addison-Wesley

[Gal] Gallmeister, Bill O.:
POSIX 4.; 1995; O'Reilly & Ass.

[Gan] Gang of Four:
Design Patterns, Elements of Reusable Object-Oriented Software; 1995; Addison-Wesley

[Gray] Gray, John Shapley:
Interprocess Communications in LINUX; 2003; Prentice Hall

[Hor] Ian Horrocks:
Constructing the User Interface with Statecharts; 1999; Addison-Wesley

[Klein] Klein, Karsten:
Grundlegende Konzepte eines Embedded Automotive Frameworks und erste Implementierung; 2005; Masterarbeit, FH-Darmstadt

[Knu] Knuth, Donald E.:
Fundamental Algorithms, 1.Vol; 1997; Addison-Wesley

[Li] Li, Qing; Yao, Caroline:
Real-Time Concepts for Embedded Systems; 2003; CMP Books

[Lin] Linden, Peter van der:
Expert C Programming; 2003; Prentice Hall

[Lip] Lippmann, Stanley B:
Essential C++; 2000; Addison-Wesley

[Mad] Maddox, Randall A.:
Distributed Application Programming in C++; 2000; Prentice Hall

[Mey] Meyers, Scott:
Effektiv C++ programmieren; 1998; Addison-Wesley

[Mey1] Meyers, Scott:
More effective C++; 2003; Addison-Wesley

[Mis] The Motor Industry Software Relieability Association:
C Guidelines; 2004

[MOST] Multimedia and Control Networking Technology, Function Catalog
MOST Cooperation, z.B. Rev 2.3, V1.0, 2001

[Nob] Noble, James; Weir, Charles:
Small Memory Software; 2001; Addison-Wesley

[Pla] Plauger, P.J.:
The standard C++ Library; 1995; Prentice Hall

[Sam] Samek, Miro:
Practical Statecharts in C/C++; 2002; CMP-Books

[Stev] Stevens, W. Richard:
UNIX Network Programming; 1999; Prentice Hall

[Schmi] Schmidt, Douglas; Stal, Michael; Rohnert, Hans; Buschmann, Frank:
Patternorientierte Software-Architektur; 2002; dpunkt.verlag

[Tan] Tanenbaum, Andrew S.:
Modern Operating Systems; 2001; Prentice Hall

[Thoe] Thoemmes, Peter:
Notizen zu C++; 2004; Springer-Verlag

[Vlis] Vlissides, John:
Pattern Hatching; 1998; Addison-Wesley

[Lee] Lee, Richard C./ Tepfenhart, William M.:
UML and C++; 2001; Prentice Hall

Index