1 CiAO Architektur

1.1 Anforderungen

1.2 Grundlegende Struktur von CiAO

CiAO ist logisch in *drei Ebenen* mit eventuellen Unterebenen organisiert. Von der Anwendung zur Hardware sind das die Ebenen *ciao*, *os*, und hw:

ciao: Diese Ebene stellt die BS- und Hardware-Zugriffsschicht aus Sicht der Anwendung da. Sie implementiert keine eigene Funktionalität, sondern enthält dünne Hüllen, die an die unteren Schichten weiterleiten. Die *ciao*-Ebene dient einzig und allein den Zweck, die Interaktionen der Anwendung mit dem BS / der Hardware durch Aspekte beeinflussen zu können.

os: Diese Ebene implementiert den BS-Kern und damit die tatsächliche Funktionalität von CiAO. Sie stellt insbesondere BS-Abstraktionen (*Thread*, *Scheduler*, *Mutex*, *MemoryManager*, ...) bereit. Ebenfalls in dieser Ebene angesiedelt sind *erweiterte Teiber*. Erweiterte Treiber sind Geräteteiber, deren Funktionalität über den reinen Hardwarezugriff hinaus geht und die deshalb mit dem BS-Kern interagieren (z.B. für Pufferverwaltung oder Freigabe der CPU während I/O Operationen).

hw: Diese Ebene stellt die Hardware einer bestimmten Plattform, insbesondere Geräte, in Form von einfachen Hardware-Zugriffsklassen bereit. Sie stellt außerdem eine Zwischenschicht für die Anbindung von Interrupts an Handler an, auch hier wieder mit dem Ziel diese durch Aspekte beeinflussen zu können.

Insgesamt ist das System also so eingekapselt, dass Kontrollflüsse sowohl „von oben“ (über Anwendungsthreads und die *ciao*-Ebene) als „von unten“ (über Interrupts und die *hw*-Ebene) durch Aspekte beeinflussbar sind.

1.2.1 Die Ebenen im Detail

Abbildung 1.1 zeigt den schematischen Aufbau sowie die Untergliederung in Subsysteme. Diese Einordnung in Ebenen und Subsysteme spiegelt sich in den zugehörigen C++ Namensräumen der Subsysteme. Der Präfix gibt die Ebene an, die optionalen weiteren Bestandteile das Subsystem. Das Suffix *std* kennzeichnet Subsysteme, die plattformübergreifenden Standardabstraktionen enthalten.

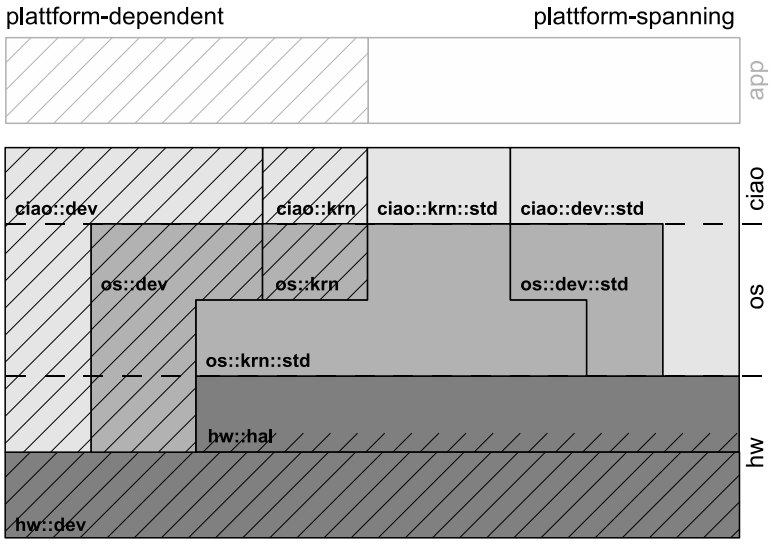


Abbildung 1.1: CiAO Ebenenmodell. Das Modell ist dreischichtig, bestehend aus den Hauptebenen *hw*, *os* und *ciao*. Plattformspezifische Teile sind gestrichelt dargestellt. Abhängigkeitsbeziehungen zwischen Subsystemen verlaufen nur von oben nach unten; wo dargestellt auch über (Unter-)Ebenen hinweg. Die Bezeichner entsprechen den zugehörigen C++ Namensräumen der Implementierung.

1.2.1.1 Ebene hw

Subsystem hw::dev **(Schnittstelle und Implementierung plattformspezifisch)**

Enthält die oben angesprochenen einfachen Hardwarezugriffsklassen, die üblicherweise plattformspezifisch sind und im wesentlichen nur Port- und Registerzugriffe kapseln. Typische Kandidaten wären z.B. `hw::dev::AVRAtmegaTC2`, `hw::dev::AVRTimer1`, `hw::dev::AVRUART0`, `hw::dev::AVRI2CChannel` auf der AVR-Plattform bzw. `hw::dev::PCKeyboard`, `hw::dev::PCTimer`, `hw::dev::CGAScreen` auf einer i386-Plattform. Enthalten sein können aber auch entsprechende Zugriffsklassen für Geräte, die auf verschiedenen Plattformen eingesetzt werden, z.B. `hw::dev::X16550UART` oder `hw::dev::HD44780LCD`.

Subsystem hw::hal **(Schnittstelle plattformübergreifend, Implementierung plattformspezifisch)**

Dieses Subsystem bietet standardisierte Abstraktionen für einen Teil der Hardware und bildet damit die Schnittstelle zwischen plattformabhängigem und plattformübergreifendem Teil von CiAO. Abstraktionen sollen zum einen für die Geräte angeboten werden, die für die Implementierung der elementaren BS-Funktionalität auf der *os*-Ebene erforderlich sind (z.B. Systemtimer oder CPU/Interrupt-Controller). Zum anderen für jene Geräte, die als *Standardgeräte* der Anwendung zugänglich gemacht werden sollen. Die Subsysteme *os::krn::std* und *os::dev::std* werden gegen *hw::hal* implementiert. Sie sind die *Referenzimplementierung* des CiAO-Kerns. Vom HAL für die Referenzimplementierung des Kerns minimal bereit gestellt werden sollen (vorläufige Liste):

- Die Klasse `hw::hal::TOC`, die den Kontextwechsel implementiert.
- Das Gerät `hw::hal::CPU`, dass insbesondere das Sperren und Freigeben von Interrupts ermöglicht (einschließlich der IRQs, die real durch einen externen Interrupt-Controller gesteuert werden).

- Das Gerät `hw::hal::SystemTimer`, das einen interruptauslösenden Zeitgeber für den Scheduler implementiert.
- Das Gerät `hw::hal::SystemClock`, das eine Systemzeit verwaltet und bereitstellt.
- Das Gerät `hw::hal::DebugPort`, das eine synchrone Zeichenausgabe für Tracing und ähnliche Zwecke bereitstellt.

Zusätzlich für Standardgeräte (mit erweiterten Treibern in *os::dev::std*, vorläufige Liste):

- Das Gerät `hw::hal::Serial0`, das eine serielle (RS232) Schnittstelle bereitstellt.
- Das Gerät `hw::hal::Timer0`, das einen weiteren interruptauslösenden Zeitgeber bereitstellt.

Das HAL abstrahiert außerdem von der Anbindung der Interrupts. Für jeden Interruptvektor gibt es eine (C++ Klasse mit statischer) Handlerfunktion, an die sich interessierte Geräte per Advice anbinden. Der eigentliche (funktionale) Teil eines IRQ-Handlers ist i.a. *nicht* Bestandteil der *hw*-Ebene, sondern liegt im CiAO-Kern oder direkt in der Anwendung. Entsprechend soll auch die Anbindung von Trapvektoren realisiert werden.

Wichtig ist, das, mit Ausnahme der Interruptanbindung, kein Teil des HALs obligatorisch ist! Ein konkretes HAL, das nur ein Subset implementiert, kann dann eben nur mit einem Teil der Referenzimplementierung zusammen verwendet werden. Das „Referenz-HAL“ ist somit eher als eine Sammlung von Geräteschnittstellen-*Konventionen* zu verstehen um den Portierungsaufwand zu begrenzen:

- Der mit dieser Konvention beschriebene, standardisierte Teil der Schnittstelle soll klein gehalten werden. Er soll also eher die Schnittmenge als die Vereinigunsmenge der Funktionalitäten der verschiedenen konkreten Geräte abbilden. Eine Funktionalität, die auf einer bestimmten Plattform nicht effizient angeboten werden kann, soll also nicht aufwändig simuliert werden.
- Die konkrete Schnittstelle darf über den Umfang der Standardschnittstelle hinaus gehen; diese erweiterte Schnittstelle jedoch nicht von der Referenzimplementierung verwendet werden. Praktisch heißt das: Ein *hw::hal*-Gerät kann ein einfacher Alias auf ein (umfangreicheres) *hw::dev*-Gerät sein, wenn dieses u.a. die vom HAL vorgeschlagene Standardschnittstelle implementiert.
- Generell sollte die Abbildung von HAL-Geräten auf konkrete Geräte konfigurierbar sein: `hw::hal::SystemTimer` könnte also z.B. entweder durch `hw::hal::AVRTimer1` oder `hw::hal::AVRAtmegaTC2` implementiert werden.

1.2.1.2 Ebene os

Subsystem os::krn::std **(Schnittstelle und Implementierung plattformübergreifend)**

Das Subsystem *os::krn::std* bietet die Standardabstraktionen des Kerns mit einer Referenzimplementierung. Damit ist insbesondere ein Thread-Konzept gemeint, also Klassen wie: `os::krn::std::Coroutine`, `os::krn::std::Thread`, `os::krn::std::Dispatcher`, `os::krn::std::Scheduler`, `os::krn::std::Mutex`, `os::krn::std::Semaphore`, `os::krn::std::MessageBox` usw.

Subsystem os::dev::std **(Schnittstelle und Implementierung plattformübergreifend)**

Dieses Subsystem bietet erweiterte Treiber für Standardgeräte, die von den Threads der Anwendung verwendet werden können. „Erweitert“ heißt hier insbesondere I/O ohne die CPU zu blockieren, d.h. die Threads warten nichtblockierend auf das Ende einer I/O Anforderung. Beispiele wären erweiterte Versionen der Standardgeräte aus dem HAL: `os::dev::std::Serial0` und `os::dev::std::Timer0`.

Des weiteren soll von diesem Subsystem eine Art Standard-I/O-Gerät / Standard-Stream angeboten werden.

Subsystem os::krn **(Schnittstelle und Implementierung plattformspezifisch)**

Dieses Subsystem enthält etwaige plattformspezifische Erweiterungen des Kerns. Falls eine Plattform z.B. einen speziellen Coprozessor mit eigenem Task-Konzept anbietet, könnte hier der zusätzlicher Scheduler dafür bereit gestellt. Ein weiteres Beispiel wären spezielle Speicherpools oder ein erweitertes Debugging-Interface.

Subsystem os::dev **(Schnittstelle und Implementierung plattformspezifisch)**

Dieses Subsystem enthält erweiterte Treiber für plattformspezifische Geräte, z.B. `os::dev::PCKeyboard`.

1.2.1.3 Ebene ciao

Die *ciao*-Ebene stellt, wie gesagt, die Schnittstelle zur Anwendung dar. Sie implementiert deshalb auch keine eigene Funktionalität, sondern wird je nach Konfiguration „angereichert“ mit Zugriffshüllen für die Abstraktionen der unteren Schichten. „Untere Schichten“ sind dabei sowohl *os* als auch *hw*. Das heißt, durch die Konfiguration der Ebenen *os* und *hw* wird festgelegt, ob die Anwendung über die *ciao*-Ebene z.B. ein bestimmtes Gerät direkt (das heißt die *hw*-Variante) oder über den erweiterten Treiber (die *os*-Variante) anspricht. Der direkte Hardwarezugriff unter Umgehung des BS ist damit grundsätzlich möglich und erwünscht.

Da die ciao-Ebene nur durch die Konfiguration der unteren Schichten eingebrachte Hüllen enthält, kann man hier nicht wirklich von einer „Implementierung“ sprechen. Dennoch muss für jedes Element eine explizite Hüllenklasse generiert oder erstellt werden, damit die gezielte Beeinflussung des Kontrollflusses durch Advice möglich ist.

Subsystem ciao::krn::std **(Schnittstelle plattformübergreifend, konfigurationsspezifisch)**

Bietet den Zugriff der Anwendung auf die Abstraktionen aus *os::krn::std*.

Subsystem ciao::dev::std **(Schnittstelle plattformübergreifend, konfigurationsspezifisch)**

Bietet den Zugriff der Anwendung auf standardisierte Geräte aus *ciao::dev::std* und/oder *hw::hal*. Außerdem wird in *ciao::dev::std* das konfigurierte Standard-I/O-Gerät eingeblendet.

Subsystem ciao::krn **(Schnittstelle plattformspezifisch, konfigurationsspezifisch)**

Bietet den Zugriff der Anwendung auf die nichtstandardisierten Kernabstraktionen aus *os::krn*.

Subsystem ciao::dev **(Schnittstelle plattformspezifisch, konfigurationsspezifisch)**

Bieted den Zugriff der Anwendung auf nichtstandardisierte Geräte aus *os::dev* und/oder *hw::dev*.

1.2.2 Konfigurationsmodelle

Das CiAO-Ebenenmodell spiegelt sich auch in der Art und Weise, wie ein CiAO-System konfiguriert werden soll. Konzeptionell erfolgt die Konfigurierung *top-down*, es werden also zunächst gewünschte Merkmale für die *os*-Ebene ausgewählt, deren Anforderungen zu (automatischen) Merkmalsselektionen der *hw*-Ebene führen. Optional können anschließend zusätzliche Merkmale der *hw*-Ebene gewählt werden bzw. eine Feinabstimmung vorgenommen werden (z.B. bei der Abbildung von *hw::hal*-Geräten auf konkrete Geräte aus *hw::dev*). Damit besteht ein CiAO-Konfigurationsraum aus üblicherweise zwei aufeinander aufbauenend Konfigurationsmodellen:

Betriebssystemmodell (ciao_os_core): Das Betriebssystemmodell bildet die Merkmale des eigentlichen Standard-CiAO-Kerns ab. Hier werden die gewünschten Merkmale für die Subsysteme *os::dev::std* und *os::krn::std* festgelegt. Außerdem sind in diesem Modell die konfigurierbaren Architekturmerkmale enthalten, deren Einfluss freilich nicht nur auf die *os*-Ebene beschränkt ist.
Das Betriebssystemmodell ist plattformübergreifend, die Merkmalsauswahl führt zu Anforderungen bezüglich der erforderlichen *hw::hal*-Merkmal, die vom Plattformmodell abgebiltet werden.

Plattformmodell (ciao_*<plattform>*): Das Plattformmodell bildet die Merkmale der Hardwareplattform ab. Hier werden die gewünschten Merkmale für die Subsysteme *hw::hal* und *hw::dev* gewählt. Eine Vorauswahl ergibt sich (automatisch) aus den Anforderungen der Betriebssystemmodellkonfiguration. Darüber hinaus können z.B. noch weitere Geräte aus *hw::hal* oder *hw::dev* gewählt werden, auf welche die Anwendung dann direkt zugreift.
Das Plattformmodell ist, wie der Name schon sagt, plattformspezifisch - auch wenn die Menge der anwählbaren *hw::hal*-Merkmale prinzipiell für jede Plattform identisch ist. Im Plattformmodell erfolgt jedoch auch die Abbildung der (abstrakten) *hw::hal*-Geräte auf die (konkreten) *hw::dev*-Geräte.

Neben diesen beiden Standardmodellen gibt es optional noch ein drittes Modell:

Betriebssystemergänzungsmodell (ciao_os-*<plattform>*): Das Betriebssystemergänzungsmodell stellt bei Bedarf die plattformspezifischen Merkmale der *os*-Ebene bereit. Das betrifft alle Merkmale aus den Subsystemen *os::dev* und *os::krn*. Die Merkmalsauswahl führt eventuell zu weiteren Anforderungen an das Betriebssystem- oder Plattformmodell.

Abbildung 1.2 zeigt noch einmal den Zusammenhang zwischen Subsystemen und Konfigurationsmodellen, sowie den möglichen Fluss von Anforderungen.

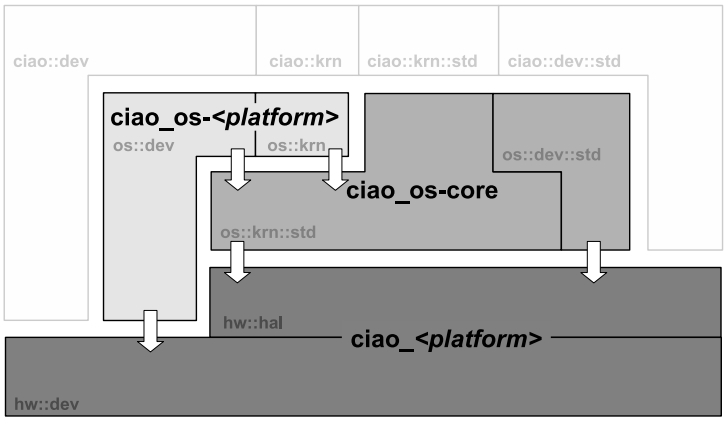


Abbildung 1.2: Zusammenhang zwischen Ebenen und Konfigurationsmodellen inklusive der möglicher Abhängigkeitsbeziehungen. Ein CiAO-Konfigurationsraum besteht aus mindestens zwei Modellen, dem hardwareabhängigen *Plattformmodell* (ciao_<*platform*>) sowie dem plattformübergreifenden *Betriebssystemmodell* (ciao_os-core). Optional kann erweiterte BS-Funktionalität über ein ergänzendes plattformspezifisches *Betriebssystemergänzungsmodell* (ciao_os-<*platform*>) bereit gestellt werden.

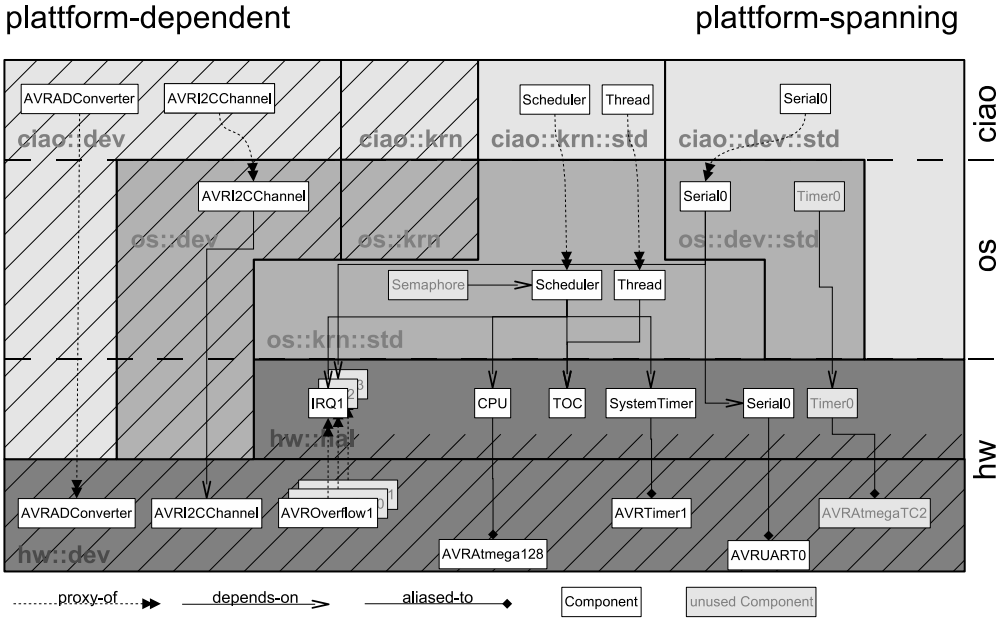
1.2.3 Ergebnis der Konfiguration

Ergebnis der Konfiguration ist ein konfigurierter Quelltextbaum für ein konkretes CiAO-Betriebssystem[1]. Technisch bedeutet dies:

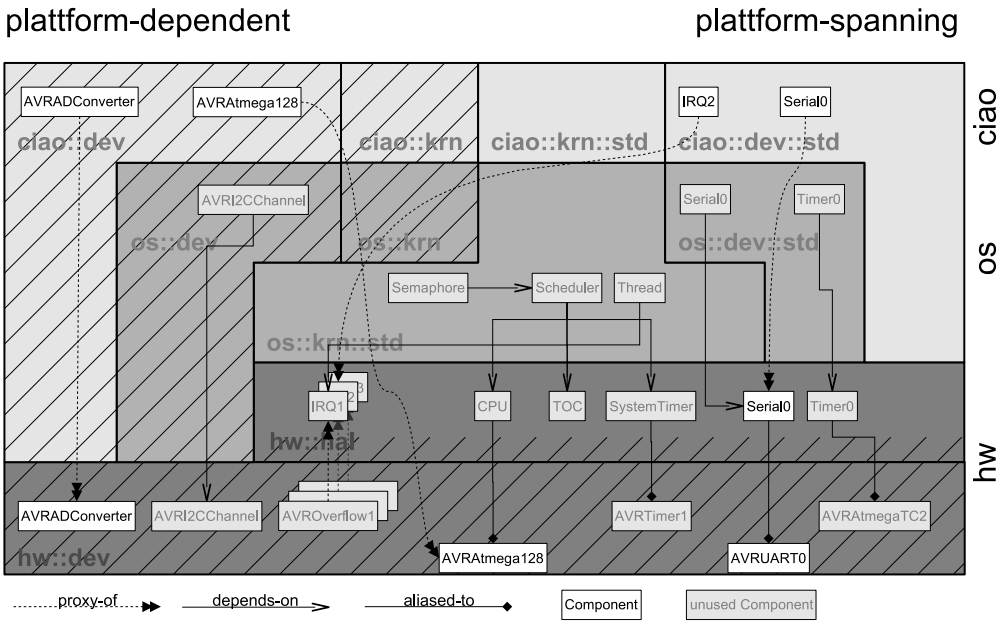
- Aus jedem Subsystem sind nur noch die Komponenten enthalten, die Merkmale implementieren, welche direkt oder indirekt (durch Abhängigkeiten) ausgewählt waren.
- Die *ciao*-Ebene enthält Proxy-Klassen für ebendiese Komponenten

Der Funktionalität („Features“) der *ciao*-Ebene wird also nicht direkt konfiguriert, sondern ergibt sich in Abhängigkeit von den ausgewählten Komponenten der tieferen Ebenen. Falls ein und dieselbe (logische) Komponente auf mehreren Ebenen vorhanden ist, so wird davon ausgegangen, dass es sich um dasselbe Gerät in verschiedenen Abstraktionsstufen handelt. In diesem Fall wird die „höchstwertige“ Version in die *ciao*-Ebene abgebildet (Abbildung 1.3)

[1] was für eine Überraschung :-)



Configuration 1



Configuration 2

Abbildung 1.3: Zwei Beispiele für Konfigurationen mit dem jeweils resultierenden Aufbau der *ciao*-Ebene. Je nach Konfiguration, werden Komponenten aus *hw* und/oder *os* in der *ciao*-Ebene bereitgestellt. Bei Komponenten, die unter gleichen Namen in verschiedenen Ebenen angeboten werden (wie z.B. `hw::hal::Serial0` und `os::dev::std::Serial0`) wird die „höchstwertige“ *selektierte* Komponente in der *ciao*-Ebene bereitgestellt.

2 Grundlegende Entwurfsprinzipien

2.1 Anforderungen

2.1.1 Das Prinzip der losen Kopplung

Ein grundlengeder Aspekt (i.w.S.)[1] nahezu aller CiAO-Entwurfsentscheidungen ist das *Prinzip der losen Kopplung*. CiAO-Komponenten sollen so entworfen sein, dass ihre Implementierung keinerlei Annahmen über andere Komponenten machen muss, solange sie diese nicht direkt benutzen. Das bedeutet insbesondere:

- Die Implementierung der Komponenten aus unteren Schichten (*hw::**) soll unabhängig davon sein, ob sie von anderen Komponenten der höheren Schichten (*os::**) *benutzt* oder *überdeckt* (bzgl. einer Abbildung in *ciao*) werden.
- Komponenten sollen sich „selbstständig“ registrieren und an allen notwendigen Stellen in das Gesamtsystem integrieren. Sozusagen „copy-plug&play“ – eine Komponente wird allein dadurch appliziert, das sie im konfigurierten Quellbaum „vorhanden ist“.

Aus *technischer Sicht* sind hier all jene Stellen von besonderem Interesse, die quasi zu einer Umkehrung der Benutztbeziehung führen:

- Alle Arten von Upcalls, z.B. vom IRQ-Handler aus der *hw::hal*-Ebene zum Gerätetreiber in die *os::dev*-Ebene.
- Die Komponenteninstantiierung. Nahezu alle Komponeten sind konzeptionell *Singletons* und müssen eine entsprechende Zugriffsoperation auf die einzige Instanz bieten. Viele Klassen in CiAO werden jedoch durch Vererbung in den höheren Schichten erweitert. In diesem Falle muss die Singletoninstanz dann logischerweise eine Instanz der erweiterten Klasse sein.
- Durch Architekturaspekte eingebrachte (und deshalb „überraschende“) *Benutztbeziehungen*. Gerätetreiber benutzten z.B. „plötzlich“ den Dispatcher, wenn IRQ-Handler auf Threads abgebildet werden.
- Eng damit im Zusammenhang steht das Problem der *Initialisierungsreihenfolge*. Die benötigte Reihenfolge der Komponenteninitialisierung beim Systemstart ist üblicherweise durch die Benutztbeziehung zwischen den Komponenten determiniert.

[1] Aspekt *im weiteren Sinne*, d.h. umgangsprachlich zu verstehen

2.1.2 Addressraumtrennung

Die *ciao*-Ebene soll die Transitionen von der Applikation in das Betriebssystem nicht nur statisch analysierbar machen, sondern es optional auch ermöglichen, dass Applikation und BS-Kern in verschiedenen Addressbereichen liegen. Dazu ist es notwendig, Code und Daten aus *ciao* optional in den Applikations-Addressbereich, die entsprechenden Elemente aus *os* und *hw* in den Kernaddressbereich legen zu können.

Aus *technischer Sicht* betrifft dies wiederum die Instantiierung der Komponenten, aber auch die Möglichkeiten des „inlining“ von Funktionen:

- Im Falle der Addressraumtrennung können die Instanzen aller Komponenten der *ciao*-Ebene *nicht* mit denen der darunterliegenden Ebene identisch sein, da sie ja in verschiedenen Addressräumen liegen sollen. Im anderen Fall sind sie entsprechend des Singleton-Prinzips jedoch identisch.
- Im Falle der Addressraumtrennung verbietet sich außerdem das einbetten von (*os*, *hw*)-Code in die *ciao*-Ebene weitestgehend, weil der *ciao*-Code im Applikationsraum liegt und somit der eingebettete Code nicht auf die (*os*, *hw*)-Zustandsvariablen zugreifen kann.

Im ersten Schritt noch nicht geplant, aber zumindestens angedacht ist außerdem die Möglichkeit, verschiedene Komponenten des Betriebssystems selber in verschiedene Adressräume aufteilen zu können. Das heißt, dass es auch für die *os*-Ebene möglich sein muss, das Einbetten von Code zu unterbinden (gleichwohl es im Normallfall ausdrücklich gewünscht ist.)

2.2 Entwurfsmuster und –idiome

2.2.1 Der kanonische Aufbau einer CiAO-Klasse

Abbildung 2.1 zeigt den prinzipiellen Aufbau einer CiAO-Klasse. Wichtig ist dabei vor allem die Singleton-Implementierung. Die statische Methode `Inst()` liefert die Referenz auf die einzige Instanz zurück, die nur über `Inst()` zugreifbar sein darf. Dadurch ist es möglich, die Instanz durch einen Aspekt auszutauschen. Des Weiteren erfolgt die Initialisierung (soweit erforderlich) nicht durch den Konstruktor, sondern durch eine Methode `init()`, die durch einen Aspekt aufgerufen wird. Dadurch ist es möglich die Initialisierungsreihenfolge konfigurierbar zu halten.

2.2.1.1 Instanzersetzung

Abbildung 2.2 zeigt das Funktionsweise der Instanzersetzung bei Erweiterung einer Singleton-Komponente durch Ableitung. Jede derartig erweitertende Komponente (hier `os::dev::std::Canonical`) bringt einen korrespondierenden *..._Inst*-Aspekt (hier `os_dev_std_Canonical_Inst`[2]) mit. Durch *around execution advice* auf die originale `Inst()`-Methode (ohne `proceed()`) wird diese unnerreichbar, was den Linker veranlasst auch die ausschließlich von dieser Methode verwendete Originalinstanz (hier `hw::hal::theCanonical_`) zu entfernen.

[2] Da Aspekte im globalen Namensraum liegen müssen, wird der Namensraum der zugehörigen Klasse (`os::dev::std`) in den Bezeichner mit eincodiert.

hw/hal/Canonical.h

```
#ifndef HW_HAL_CANONICAL_H
#define HW_HAL_CANONICAL_H

namespace hw { namespace hal {

class Canonical {
  static Canonical theCanonical_; // the one-and-only instance of hw::hal::Canonical
                                  // theCanonical_ is *only* accessed by Inst()

  void init();                    // [OPTIONAL] startup time initialization function.
                                  // only called by a corresponding _Init aspect

  protected:
  Canonical() {}                  // constructor, must be declared protected,
                                  // must be defined, but may be empty.
                                  // Initalization itself is done in Init()
  public:
  static Canonical& Inst() {      // the singleton access function
    return theCanonical_;
  }

  void reset();                   // [OPTIONAL] user callable (re-)initialization function

  // selectors, modifiers, operations
  void operation_A( int );

};

}}  // namespace hw::hal

#endif // HW_HAL_CANONICAL
```

Abbildung 2.1: Kanonischer Aufbau einer CiAO-Klasse. Die Klasse `hw::hal::Canonical` ist ein *Singleton*, dessen einzige Instanz `theCanonical_` nur über die Methode `Canonical& Canonical::Inst()` zugreifbar sein darf. Um jede anderweitige Instantiierung zu unterbinden muss ein *protected* Konstruktor definiert werden. Die eigentliche Initialisierung (falls erforderlich) findet jedoch in der Methode `Canonical::init()` statt.

2.2.1.2 Initialisierung

Komponenten, die bereits beim Systemstart initialisiert werden müssen, verfügen über eine private `init()`-Methode. Diese dient ausschließlich der Initialisierung beim Systemstart; eine eventuelle Rücksetzmöglichkeit durch das Benutzerprogramm erfolgt in der optionalen Methode `reset()`.

Die `init()`-Methode wird aktiviert durch einen korrespondierenden ..._Init*-Aspekt, der sich an die globale `init()`-Funktion der jeweiligen Ebene bindet. Die Bindung durch Aspekte ermöglicht es mit entsprechenden order-advice auf sehr einfache Art und Weise Abhängigkeiten bei Initialisierungsreihenfolge zu modellieren. Abbildung 2.3 zeigt dieses Modell an einem Beispiel

2.2.2 Konfiguration der *ciao*-Ebene

Die *ciao*-Ebene stellt die Schnittstelle zur Anwendung dar. Wie in den Abschnitten 1.2 bzw. 1.2.1.3 dargelegt, werden die konfigurierten *hw*- und *os*-Komponenten mit ihrer jeweils „höchstwertigen“ Implementierung in die *ciao*-Ebene gespiegelt. Neben der Auswahl der „höchstwertigen“ Implementierung müssen weitere Konfigurationsentscheidungen berücksichtigt werden, insbesondere die Frage der Adressraumtrennung, die Einfluss auf die Instanzerzeugung und das Einbetten von Funktionen hat.

Für jede von der Anwendung zu benutzenden Komponente ist also eine entsprechende Zugriffsklasse in die *ciao*-Ebene einzufügen. Diese *Proxyklassen* können und sollen auf lange Sicht automatisch generiert

os/dev/std/Canonical.h

```
#ifndef OS_DEV_STD_CANONICAL_INST_H
#define OS_DEV_STD_CANONICAL_INST_H

#include "hw/hal/Canonical.h"

namespace os { namespace dev { namespace std

class Canonical : public hw::hal::Canonical {

  static Canonical theCanonical_; // the one-and-only instance of os::dev:std::Canonical
                                  // (as well as the *new* one-and-only instance of hw::hal::Canonical)

  void init();                    // [OPTIONAL] startup time initialization function.
                                  // only called by a corresponding _Init aspect


  protected:
  Canonical() {}
  public:
  static Canonical& Inst() {
    return theCanonical_;
  }

  // new selectors, modifiers, operations
  int operation_B();
};

}}  // namespace os::dev::std
#endif // OS_DEV_STD_CANONICAL_INST_H
```

os/dev/std/Canonical_Inst.ah

```
#ifndef OS_DEV_STD_CANONICAL_INST_AH
#define OS_DEV_STD_CANONICAL_INST_AH

#include "hw/hal/Canonical.h"

aspect os_dev_std_Canonical_Inst {
  advice execution( "% ...::Inst()" )
  && within( base( "os::dev::::std::Canonical") ) : around() {
    *tjp->result() = &os::dev::std::Inst();
  }
};

#endif // OS_DEV_STD_CANONICAL_INST_AH
```

Abbildung 2.2: Instanzersetzung bei Verfeinerung. Die Klasse `os::dev::std::Canonical` ist eine Verfeinerung von `hw::hal::Canonical` (Abbildung 2.1). Die Originalinstanz **`hw::hal::`**`theCanonical_` wird über den zu `os::dev::std::Canonical` gehörenden Aspekt `os_dev_std_Canonical_`**`Inst`** mit **`os::dev::std::`**`theCanonical_` substitutiert.

werden. Ein entsprechendes Tool muss jedoch noch etwickelt werden. Zunächst ist daher (auch zur Evaluierung des Ansatzes) eine „händische“ Implementierung der *ciao*-Klassen vorgesehen. Eine solche „händische“ Implementierung muss, gegeben durch die vielfältigen Konfigurationen, in hohem Maße bedingte Kompilierung und andere Hässlichkeiten verwenden. Im Folgenden werden daher zunächst diese zu berücksichtigen Konfigurationsmöglichkeiten erläutert. Anschließend werden der Aufbau einer *ciao*-Klasse sowie ein Implementierungschema für die händische Implementierung beschrieben.

hw/hal/Canonical_Init.ah

```
#ifndef HW_HAL_CANONICAL_INIT_AH
#define HW_HAL_CANONICAL_INIT_AH

#include "hw/hal/Canonical.h"
aspect hw_hal_Canonical_Init {
  advice execution( "% hw::init()" ) : before() {
    hw::hal::Canonical::Inst().init();
  }

  // [OPTIONAL] define initialisation order dependencies
  // (e.g. ensure that Other is initialized before me)
  advice execution( "% hw::init()") : order( "hw_hal_Other_Init","hw_hal_Canonical_Init" )
};

#endif // HW_HAL_CANONICAL_INIT_AH
```

Abbildung 2.3: *_Init*-Aspekt für die Komponente `hw::hal::Canonical`. Der Aufruf der Initialisierungsmethode wird an die globale Initalisierungsfunktion der Ebene gebunden, wobei durch *order*-advice sichergestellt wird, dass alle Komponenten von denen `hw::hal::Canonical` abhängt (hier `hw::hal::Other`) vorher initialisiert werden.

2.2.2.1 Globale Konfigurationsentscheidungen

Die Konfiguration erfolgt mit pure::variants. Für die händische Implementierung der *ciao*-Ebene werden die relevanten Konfigurationsentscheidungen über Präprozessorsymbole bereitgestellt. Folgende Symbole definieren die grundsätzliche Konfiguration der *ciao*-Ebene:

cfCIAO_DECOUPLE: Die Instanzen der *ciao*-Ebene sollen von den eigentlichen Implementierungen aus der *os*- bzw. *hw*-Ebene entkoppelt werden, um z.B. eine Adressraumtrennung zwischen der Anwendung (welche die *ciao*-Instanzen enthält) und dem eigentlichen BS-Kern durchzusetzen. Dieses bedeutet insbesondere:

- Die (Singleton-)Instanzen der ciao-Klassen dürfen **nicht** die jeweilige Instanz der Implementierung ersetzen, was im anderen Fall jedoch erforderlich ist.
- Die *ciao*-Klassen können nicht von der jeweiligen Implementierungsklasse erben. Stattdessen müssen sie eine (untypisierte) Referenz auf die tatsächliche Instanz der Implementierungsklasse beinhalten (Aggregation statt Implementierungsvererbung).
- Der *ciao*-Code darf nicht eingebettet werden (inline)[3].

cfCIAO_INLINE: Die Implementierung der *ciao*-Klassen soll in die Anwendung eingebettet werden (inline). Diese Konfigurationsoption schließt die Option cfCIAO_DECOUPLE aus.

Zusätzlich gibt es noch eine weitere Option, die nur indirekt auf die Klassen der *ciao*-Ebene Einfluss hat:

[3] Grund dafür ist die Implementierung der Adressraumtrennung, die sich relativ einfach implementieren ließe wenn man garantieren kann, dass *ciao*-Code/Instanzen physisch in einem wohldefinierten Segment liegen. Die Idee ist, drei Adressbereiche vorzusehen: a) den Kernadressbereich, der von der Anwendung weder gelesen noch beschrieben werden kann, b) den Anwendungsadressbereich, der von der Anwendung sowohl gelesen als auch beschrieben werden kann, und als Bindeglied c) den Anwendungsaddressbereich für die Komponenten aus der *ciao*-Ebene, der von der Anwendung nur gelesen werden kann und die Traps in den Kern durchführt. Dadurch, dass die Funktionen und Instanzen der ciao-Ebene nicht kompromittiert werden können, reicht zur Validierung eines Traps auf Kernseite ein einfacher Test aus, welcher anhand der Rücksprungadresse überprüft, ob der Trap aus dem *ciao*-Segment kommt. Die Konzeptstudie *protection* (`concepts/protection`) demonstriert diese Vorgehensweise.

cfOS_INLINE: Die Implementierung der *os*-Klassen soll vollständig (in die *ciao*-Ebene) eingebettet werden. Dieses ist vor allem im Zusammenhang mit cfCIAO_DECOUPLE sinnvoll, da so eine doppelte Indirektion verhindert werden kann. Umgekehrt kann es aber auch sinnvoll sein, die größeren Funktionen der *os*-Ebene *nicht* einzubetten, z.B. wenn cfCIAO_INLINE gesetzt ist und die Anwendung sehr viele Systemaufrufe tätigt.

2.2.2.2 Komponentenspezifische Konfigurationsentscheidungen

Neben den oben beschriebenen globalen Konfigurationsentscheidungen gilt es für viele Komponenten der *ciao*-Ebene noch festzulegen, ob ihre *hw*- oder ihre *os*-Implementierung (falls vorhanden) abgebildet werden soll. Zu diesem Zweck sollen für die händische Implementierung weitere Flags von pure::variants generiert werden, die für die kanonische CiAO-Komponente `Canonical` folgenden Aufbau hätten:

cfCIAO_CANONICAL_IMPL_HW: Die höchstwertige Version von `Canonical`, die für das Zielsystem konfiguriert wurde, ist die aus der *hw*-Schicht (hier `hw::hal::Canonical`). Für diese soll ein Proxy in der *ciao*-Ebene erstellt werden (hier `ciao::os::dev::std::Canonical`).

cfCIAO_CANONICAL_IMPL_OS: Die höchstwertige Version von Canonical, die in das Zielsystem konfiguriert wurde, ist die aus der *os*-Ebene (hier `os::dev::std::Canonical`). Für ebendiese soll der Proxy in der *ciao*-Ebene erstellt werden (hier `ciao::os::dev::std::Canonical`).

2.2.3 Komponentenimplementierung für die *ciao*-Ebene

Bei der Implementierung der *ciao*-Komponenten wird man üblicherweise *eine* Implementierung für alle Konfigurationen vorsehen, da die Gemeinsamkeiten der verschiedenen Konfigurationen ingesamt überwiegen. Das bedeutet jedoch auch die Verwendung bedingter Kompilierung, sowie extra Implementierungsdateien für die Inline-Funktionen. Die folgenden Abschnitte schildern, am Beispiel der kanonischen CiAO-Komponente, wie die Konfigurationsflags entsprechend umgesetzt werden können.

2.2.3.1 Durchsetzen der komponentenspezifischen Konfigurationsentscheidungen

Je nachdem ob cfCIAO_CANONICAL_IMPL_HW oder cfCIAO_CANONICAL_IMPL_OS gesetzt sind, muss `hw::hal::Canonical` oder `os::dev::std::Canonical` als Implementierungsklasse verwendet werden, die wiederum in unterschiedlichen Header-Dateien deklariert sind. Diese werden zunächst ermittelt und über zwei weitere Makros zur Verfügung gestellt[4]:

```
#ifdef cfCIAO_CANONICAL_IMPL_HW
#  define cfCIAO_CANONICAL_IMPL hw::hal::Canonical
#  define cfCIAO_CANONICAL_IMPL_HEADER "hw/hal/Canonical.h"
#else
#  define cfCIAO_CANONICAL_IMPL os::dev::std::Canonical
#  define cfCIAO_CANONICAL_IMPL_HEADER "os/dev/std/Canonical.h"
#endif
```

[4]Diese Macros könnten alternativ auch direkt von pure::variants generiert werden.

2.2.3.2 Durchsetzen der globalen Konfigurationsentscheidungen

Wenn cfCIAO_DECOUPLE definiert ist, muss die eigentliche Implementierung vor der Anwendung verborgen bleiben. Dazu darf die *ciao*-Klasse nicht von der sie implementierenden Klasse erben, sondern muss über eine Referenz mit ebendieser verbunden werden. Im anderen Fall (cfCIAO_DECOUPLE nicht definiert) muss jedoch die unter 2.2.1.1 geschilderte Instanzersetzung bzgl. der (Singleton-)Instanz der implementierenden Klasse durchgeführt werden, d.h. die *ciao*-Klasse muss von der implementierenden Klasse erben.
Die Methodenimplementierungen, die ja faktisch nichts weiter tun als an die entsprechende Methode aus der implementierenden Klasse weiterzuleiten, müssen unabhängig davon sein, ob Vererbung oder Assoziation verwendet wird. Um dieses zu erreichen, wird auf die Instanz der implementierenden Klasse nur über eine eigene Methode `Impl* impl()` zugegriffen, die im Falle von Vererbung schlichtweg die eigene Instanz zurückgibt, im Falle von Assoziation die Instanz der implementierenden Klasse:

```
#ifndef cfCIAO_DECOUPLE
#  include cfCIAO_CANONICAL_IMPL_HEADER
#endif
namespace ciao { namespace dev { namespace std {

#ifndef cfCIAO_DECOUPLE
class Canonical : public cfCIAO_CANONICAL_IMPL {
  typedef cfCIAO_CANONICAL_IMPL Impl;
  Impl* impl() { return this; }
  const Impl* impl() const { return this; }
#else
class Canonical {            // no superclass
  struct Impl;               // forward declare Impl only (as struct)
  Impl* impl();
  const Impl* impl() const;
#endif
  ...                        // selectors, modifiers, operations
 }; } } }
```

In der Implementierungsdatei erfolgt dann über `impl()` die Weiterleitung an die implementierende Klasse. Eventuell wird auch erst hier die Headerdatei der implementierenden Klasse eingebunden. Um das optionale *Inlining* zu ermöglichen, steht die Implementierung zudem in einer eigenen .inl-Datei, die je nach Konfiguration von cfCIAO_INLINE in die .h/.cpp-Datei eingebunden wird:

```
#include cfCIAO_CANONICAL_IMPL_HEADER  // may not have been included yet
#ifdef cfCIAO_INLINE                   // methods shall be declared as inline
#  define _INLINE inline
#else
#  define _INLINE
#endif

namespace ciao { namespace dev { namespace std {
#ifdef cfCIAO_DECOUPLE
  // define forward declared Impl type and accessor functions
  struct Canonical::Impl : public cfCANONICAL_IMPL {};
  inline Canonical::Impl* CLASS::impl()
        { return static_cast< Impl* >( &Impl::Inst() ); }
  inline const CLASS::Impl* CLASS::impl() const
        { return static_cast< const Impl* >(&Impl::Inst() ); }
```

```
#endif

// implementation uses impl()
_INLINE  void Canonical::operation_A( int n ) {
  impl()->operation_A( n );
}
...
} } }
```

Im Falle von cfCIAO_DECOUPLE wird hier auch der forward-deklarierte Typ `Impl` definiert[5].

Wie bei den unterliegenden Ebenen, gilt auch für die Komponenten der *ciao*-Ebene, dass sie *Singleton*-Instanzen bilden. Die Implementierung der Singleton-Eigenschaft erfolgt dabei genau so, wie in Abschnitt 2.2.1.1 beschrieben, wobei die Instanzersetzung durch den zugehörigen `_Inst`-Aspekt natürlich nur erfolgen darf, wenn cfCIAO_DECOUPLE nicht definiert ist. Dieses Anforderung ist bei dem in 2.2.1.1 beschrieben Implementierungsschema implizit gegeben durch das *Matching* über die Basisklasse, die es ja nur in diesem Fall gibt.

2.2.3.3 Zusammenfassendes Beispiel

Abbildung 2.4 zeigt noch mal im Zusammenhang und am Beispiel der kanonischen Komponente, wie eine händische Implementierung der *ciao*-Ebene aussieht. Die Implementierung der Singleton-Eigenschaft und der `impl()`-Methode wurde dabei in wiederverwendbare Makros ausgelagert, deren Definition in Abbildung 2.5 zu finden ist[6].

2.3 Programmierkonventionen

[5] Der Trick über die Deklaration/Definition einer `struct`, die vom Implementierungstypen *erbt* ist erforderlich, da der reale Implementierungstyp nicht so direkt forward-deklariert werden kann. Der Grund ist, dass der genaue Namensraum in dem dieser Typ liegt unbekannt ist. Eine Forward-Deklaration der Art `class A::B::C;` schlägt fehl, wenn `A` und `B` nicht bekannt sind, da der Compiler nicht davon ausgehen kann, dass es sich bei `A` und `B` um Namensräume handelt. Stattdessen müsste man also schreiben: `namespace A{namespace B{class C;}}`. Nur lässt sich das dann schlecht konfigurieren, da man ein Präprozessormakro wie cfCIAO_CANONICAL_IMPL nicht in ebendiese Bestandteile zerlegen kann.

[6] Ein ablauffähiges Beispiel, welches diese Konzepte verdeutlicht findet sich auch in der Konzeptstudie *instantiation* (`concepts/instantiation`).

a) ciao/dev/std/Canonical.h

```
#ifndef CIAO_CANONICAL_H
#define CIAO_CANONICAL_H

#include "macros.h"

#ifdef cfCIAO_CANONICAL_IMPL_HW
# define cfCIAO_CANONICAL_IMPL hw::hal::Canonical
# define cfCIAO_CANONICAL_IMPL_HEADER "hw/hal/Canonical.h"
#else
# define cfCIAO_CANONICAL_IMPL os::dev::std::Canonical
# define cfCIAO_CANONICAL_IMPL_HEADER "os/dev/std/Canonical.h"
#endif

#ifndef cfCIAO_DECOUPLE
# include cfCIAO_CANONICAL_IMPL_HEADER
#endif

namespace ciao { namespace dev { namespace std {

class CIAO_CLASS( Canonical, cfCIAO_CANONICAL_IMPL) {
  CIAO_CLASS_BODY( Canonical, cfCIAO_CANONICAL_IMPL );
public:
// methods defined by both, os::dev::std::Canonical
// and hw::hal::Canonical
  void operation_A( int );

// additional methods, only defined by os::dev::std::Canonical
#ifndef cfCIAO_CANONICAL_IMPL_HW
  int operation_B();
#endif

};

}}} // namespace ciao::dev::std

#ifdef cfCIAO_INLINE        // if ciao-layer shall be inlined
# include "Canonical.inl" // include implementation here
#endif
#endif // CIAO_CANONICAL_H
```

b) ciao/dev/std/Canonical.cpp

```
#include "Canonical.h"

#ifndef cfCIAO_INLINE
  #include "Canonical.inl"
#endif

// the one and only Canonical singleton instance
ciao::dev::std::Canonical
    ciao::dev::std::Canonical::theCanonical_;
```

c) ciao/dev/std/Canonical.inl

```
#include "Canonical.h"
#include cfCIAO_CANONICAL_IMPL_HEADER

#ifdef cfCIAO_INLINE    // methods shall be declared as inline
# define _INLINE inline // only if ciao-layer itself is inlined
#else
# define _INLINE
#endif

namespace ciao { namespace dev { namespace std {

// Define standard stuff (Inst(), Impl, impl())
CIAO_CLASS_DEF( Canonical, cfCIAO_CANONICAL_IMPL );

_INLINE void Canonical::operation_A( int n ) {
  impl()->operation_A( n );
}

// additional methods, only defined by os::dev::std::Canonical
#ifndef cfCIAO_CANONICAL_IMPL_HW

_INLINE int Canonical::operation_B() {
  return impl()->operation_B();
}

#endif // cfCIAO_CANONICAL_IMPL_HW

}}} // namespace ciao::dev::std

#undef _INLINE
```

d) ciao/dev/std/Canonical_Inst.ah

```
#ifndef CIAO_DEV_STD_CANONICAL_INST_AH
#define CIAO_DEV_STD_CANONICAL_INST_AH

#include "Canonical.h"

aspect ciao_dev_std_Canonical_Inst {

  advice execution( "% ...::Inst()" )
    && within( base( "ciao::dev::std::Canonical" ) ) : around() {
    *tjp->result() = &ciao::dev::std::Canonical::Inst();
  }
};


#endif // CIAO_DEV_STD_CANONICAL_INSTG_AH
```

Abbildung 2.4: Implementierung der Abbildung einer kanonischen CiAO-Komponente in die *ciao*-Ebene. Erforderlich sind üblicherweise 4 Dateien: Headerdatei (**a**), Übersetzungseinheit (**b**), Implementierungsdatei (**c**) und Instanzersetzungsaspekt (**d**). In Abhängigkeit von cfCIAO_INLINE, wird die Implementierungsdatei entweder am Ende des Headers oder zu Beginn der Übersetzungseinheit eingebunden. Drei spezielle Präprozessormakros (CIAO_CLASS, CIAO_CLASS_BODY und CIAO_CLASS_DEF, siehe Abbildung 2.5) fügen die jeweils konfigurationsabhängigen Deklarationen und Definitionen von `Inst()`, `Impl` und `impl()` ein.

macros.h

```
#ifndef __MACROS_H__
#define __MACROS_H__

// declares/defines the singleton stuff in the class body
// note that the instance itself has to be defined in a CPP file
#define CIAO_SINGLETON(CLASS)\
  private:\
    static CLASS the##CLASS##_;\
  protected:\
    CLASS(){}\
  public:\
    static CLASS& Inst(){ return the##CLASS##_;}

#ifdef cfCIAO_DECOUPLE
# define CIAO_CLASS(CLASS, IMPL) CLASS
# define CIAO_CLASS_BODY(CLASS, IMPL)\
    struct Impl;\
    Impl* impl();\
    const Impl* impl() const;\
    CIAO_SINGLETON(CLASS)

# define CIAO_CLASS_DEF(CLASS, IMPL) \
    struct CLASS::Impl : public IMPL {};\
    inline CLASS::Impl* CLASS::impl() { return static_cast< Impl* >( &Impl::Inst() ); }\
    inline const CLASS::Impl* CLASS::impl() const { return static_cast< const Impl* >( &Impl::Inst() ); }

#else // cfCIAO_DECOUPLE

# define CIAO_CLASS(CLASS, IMPL) CLASS : public IMPL
# define CIAO_CLASS_BODY(CLASS, IMPL)\
    typedef IMPL Impl;\
    Impl* impl() { return this; }\
    const Impl* impl() const { return this; }\
    CIAO_SINGLETON(CLASS)

# define CIAO_CLASS_DEF(CLASS, IMPL)

#endif // cfCIAO_DECOUPLE

#endif // __MACROS_H__
```

Abbildung 2.5: Hilfsmakros zur händischen Implementierung der *ciao*-Ebene

Bezeichnerart	Schreibweise	Beispiele
globale Variablen funktionslokale statische variablen, statische Membervariablen	großer Anfangsbuchstabe, Trennung durch mixed-case, Hauptwörter, Präfix **`the`**. Statische Membervariablen erhalten zuätzlich einen Unterstrich als Suffix	`theApplication`, `theRCBuffer`, `theSerial0_`
Membervariablen	klein, Trennung durch Unterstrich, Unterstrich als Suffix	`id_. last_value_`
Funktionen/Methoden	kleiner Anfangsbuchstabe, Trennung durch mixed-case	`getValue()`, `startup()`, `currentThread()`
Wert-Konstanten **`enum`**/**`const`** -Werte, Makros für Wertkonstanten	groß, Trennung durch Unterstrich	`TIMER_IOPORT, STACK_MAX`
Flag-Konstanten Konstanten für kombinierbare Flags	wie Konstanten, mit Suffix **`_F`**	`USEIRQ_F`, `ONE_STOPBIT_F`, `XONXOFF_F`
Makros funktionale, code-generierende Makros	groß, Trennung durch Unterstrich, Präfix **`CIAO`**_	`CIAO_CLASS_BODY()`, `CIAO_CLASS()`
Konfigurationseinstellungen Makros zur Konfiguration, üblicherweise von pure::variants generiert	groß, Trennung durch Unterstrich, Präfix **`cf`**	`cfCIAO_DECOUPLE`, `cfTASKS_MAX`
Namensräume	klein	`os, ciao, std, hw`
Typen Klassen, Basisdatentypen, …	großer Anfangsbuchstabe, Trennung durch mixed-case, Hauptwörter	`Serial0, UInt32, Int8, AVRUART0`
Typschablonen Templates/Generatoren, deren Ergebnis ein instantiierbarer Typ ist	klein, Trennung durch Unterstrich Hauptwörter mit Suffix **`_t`**, optional ergänzt durch weitere qualifizierende Suffixe	`int_t<…>, int_t_min<…>, vector<…>`
Aspekte	wie Typen, wobei der logische Namensraum als Präfix und ggfs. eine etwaige Standardfunktion als Suffix dazu kommen können. Trennung der Präfixe/Suffixe durch _	`ciao_Init`, `os_dev_std_Serial0_Inst`
Pointcuts	wie Typen, jedoch mit Präfix **`pc`**	`pcKernel()`, `pcSynchronized()`

Tabelle 2.1: Namenskonventionen für Bezeichner