

	OSAL
Inhalt	1
1 Inhalt	4
2 Typisches Software-Schichtenmodell mit einer Betriebssystem-Abstraktionsschicht	5
3 Betriebssystem-Abstraktionsschicht	7
3.1 Vorteile und Nachteile.....	7
3.2 OS-Mechanismen.....	8
3.2.1 Übersicht.....	8
3.2.2 Objektorientierter Ansatz.....	10
4 Task-Abstraktion.....	11
4.1 FreeRTOS API Übersicht.....	11
4.2 Lösung.....	12
4.3 Task-Erzeugung	13
4.3.1 Anwendung mit Task-Mehrfach-Instanziierung.....	13
4.3.2 Manuelle Übergabe des this-Pointers.....	14
4.4 Task-Terminierung.....	15
4.4.1 Einfacher Wrapper	16
5 Mailbox-Abstraktion	17
5.1 FreeRTOS API Übersicht.....	17
5.2 Lösung.....	18
5.3 Template-Klasse.....	19
5.4 Mailbox-Erzeugung.....	20
5.4.1 Konstruktor versus eigenständige Operation.....	20
5.4.2 Anwendung.....	21
6 Weitere Abstraktionskonzepte mit C++	22

6.1	Diskussionspunkte	22
6.2	Quellen standardisierter Interfaces für eine Betriebsabstraktion.....	23
7	Resümee.....	24

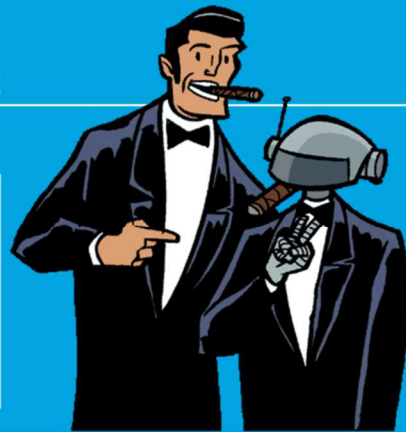
Vortragsreihe Echtzeit – Embedded Software Engineering Kongress 2014



**Programmierung einer
Betriebssystem-Abstraktionsschicht**
Konzepte und Umsetzung mit C++

MICROCONSULT GmbH

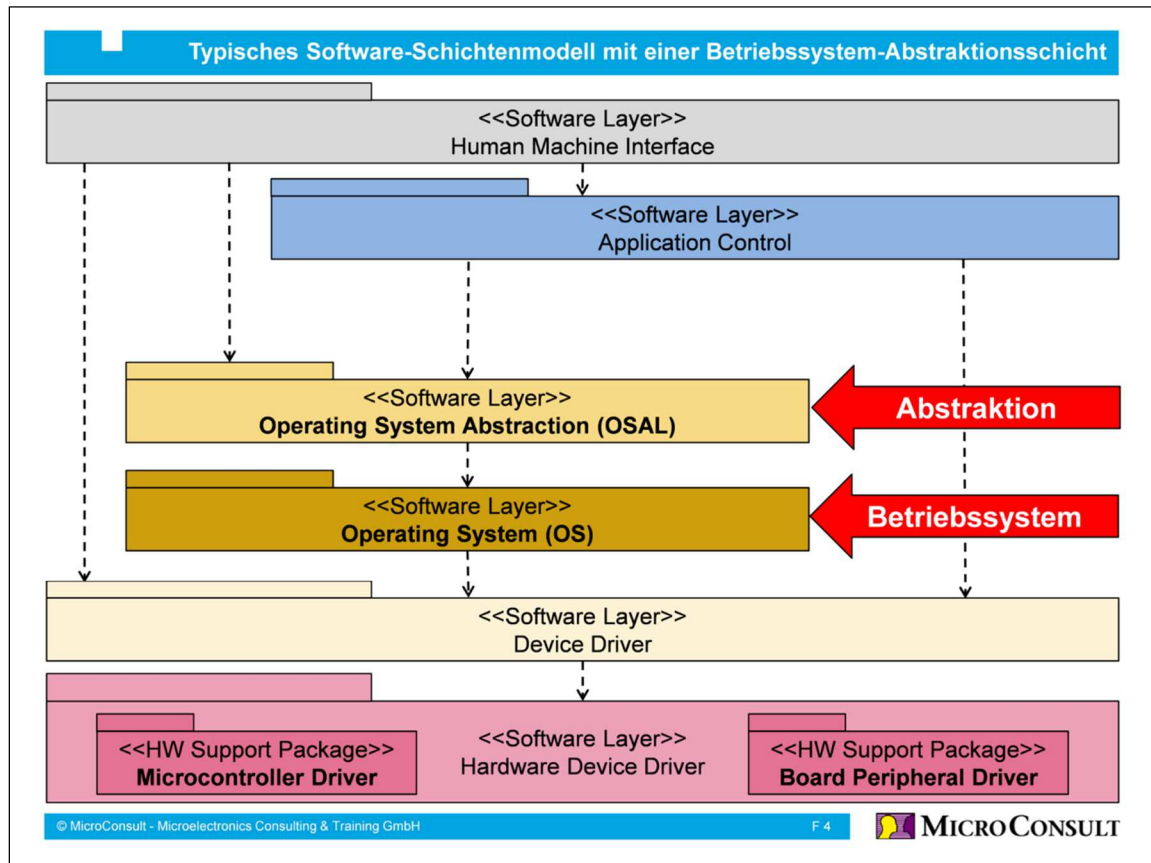
Dipl.-Ing. (FH) **Thomas Batt**
Manager Management-Training und -Coaching
Trainer & Coach für Embedded- und Echtzeitsysteme
Charles-de-Gaulle-Str. 6 • 81737 München • Germany
Tel.: +49 (0)89 450617-35
FAX: +49 (0)89 450617-17
E-Mail: t.batt@microconsult.com
Internet: www.microconsult.de



1 Inhalt

Inhalt
▪ Typische Software-Architektur mit einer Betriebssystem-Abstraktionsschicht
▪ Vor- und Nachteile beim Einsatz einer Betriebssystem-Abstraktionsschicht
▪ OS-Mechanismen in der Übersicht
▪ OS-Mechanismen und objektorientierter Ansatz
▪ C++ Abstraktionsbeispiel: Task
▪ C++ Abstraktionsbeispiel: Mailbox
▪ Weitere Abstraktionskonzepte mit C++
▪ Quellen standardisierter Interfaces für eine Betriebssystem-Abstraktion
▪ Resümee
Download-Link für diese Präsentation: http://download.microconsult.net/ese2014/osal.zip
© MicroConsult - Microelectronics Consulting & Training GmbH 10.11.2014 F 3  MICROCONSULT

2 Typisches Software-Schichtenmodell mit einer Betriebssystem-Abstraktionsschicht



Muss in einer Software-Architektur das Betriebssystem getauscht werden, gibt es prinzipiell drei verschiedene Lösungsszenarien:

1. Direkter Austausch des Betriebssystems (keine Betriebssystem-Abstraktion vorhanden)

Konsequenzen:

- Verfügbarkeit und Arbeitsweise der Betriebssystem-Mechanismen zwischen alt und neu vergleichen
- Alle Betriebssystem-Aufrufe in jeder zugreifenden Schicht auf die neuen Aufrufe anpassen (HOHER AUFWAND!)
- Umsetzung testen

Fazit:

Bei erneutem Betriebssystem-Austausch unveränderte Konsequenzen und damit hoher Aufwand.

2. Das zu ersetzende Betriebssystem mutiert zur Betriebssystem-Abstraktion (noch keine Betriebssystem-Abstraktion vorhanden)

Konsequenzen:

Typisches Software-Schichtenmodell mit einer Betriebssystem-Abstraktionsschicht

- Verfügbarkeit und Arbeitsweise der Betriebssystem-Mechanismen zwischen alt und neu vergleichen
- Die Betriebssystem-Aufrufe des alten Betriebssystems als Betriebssystem-Abstraktion verwenden
- Die Aufrufe des alten Betriebssystems auf die des neuen in der Betriebssystem-Abstraktion umsetzen
- Umsetzung testen

Fazit:

Schnelle Lösung, um eine Betriebssystem-Abstraktion zu erhalten, die aber wenig ausgereift und überlegt ist. Aber wie sieht es beim nächsten Betriebssystemwechsel aus?

3. Neu entwickelte Betriebssystem-Abstraktionsschicht

Konsequenzen:

- Verfügbarkeit und Arbeitsweise der Betriebssystem-Mechanismen zwischen alt und neu vergleichen
- Eine zukunftssichere Betriebssystem-Abstraktionsschicht entwickeln
- Alle Betriebssystem-Aufrufe in jeder zugreifenden Schicht auf die neuen Aufrufe anpassen (HOHER AUFWAND!)
- Die Betriebssystem-Abstraktionsschicht auf das neue Betriebssystem adaptieren
- (Betriebssystem-Aufrufe des alten Betriebssystems als Betriebssystem-Abstraktion verwenden
- Umsetzung testen
- Zur Überprüfung, die Betriebssystem-Abstraktion eventuell noch auf ein zweites Betriebssystem portieren

Fazit:

Initial aufwendigste Lösung, rechnet sich aber mit jedem zukünftigen Betriebssystem-Wechsel.

➔ **Direkt bei jeder neuen Software-Architektur Entwicklung berücksichtigen**

3 Betriebssystem-Abstraktionsschicht

3.1 Vorteile und Nachteile

Betriebssystem-Abstraktionsschicht – Vorteile und Nachteile

Vorteile bei Verwendung einer Betriebssystem-Abstraktionsschicht:

- Hoher Entkopplungsgrad zwischen Betriebssystem und der Applikation
 - ➔ Schnelle Austauschbarkeit des Betriebssystems
 - ➔ Schnelle Portierbarkeit der Applikation auf ein anderes Betriebssystem

Nachteile bei Verwendung einer Betriebssystem-Abstraktionsschicht:

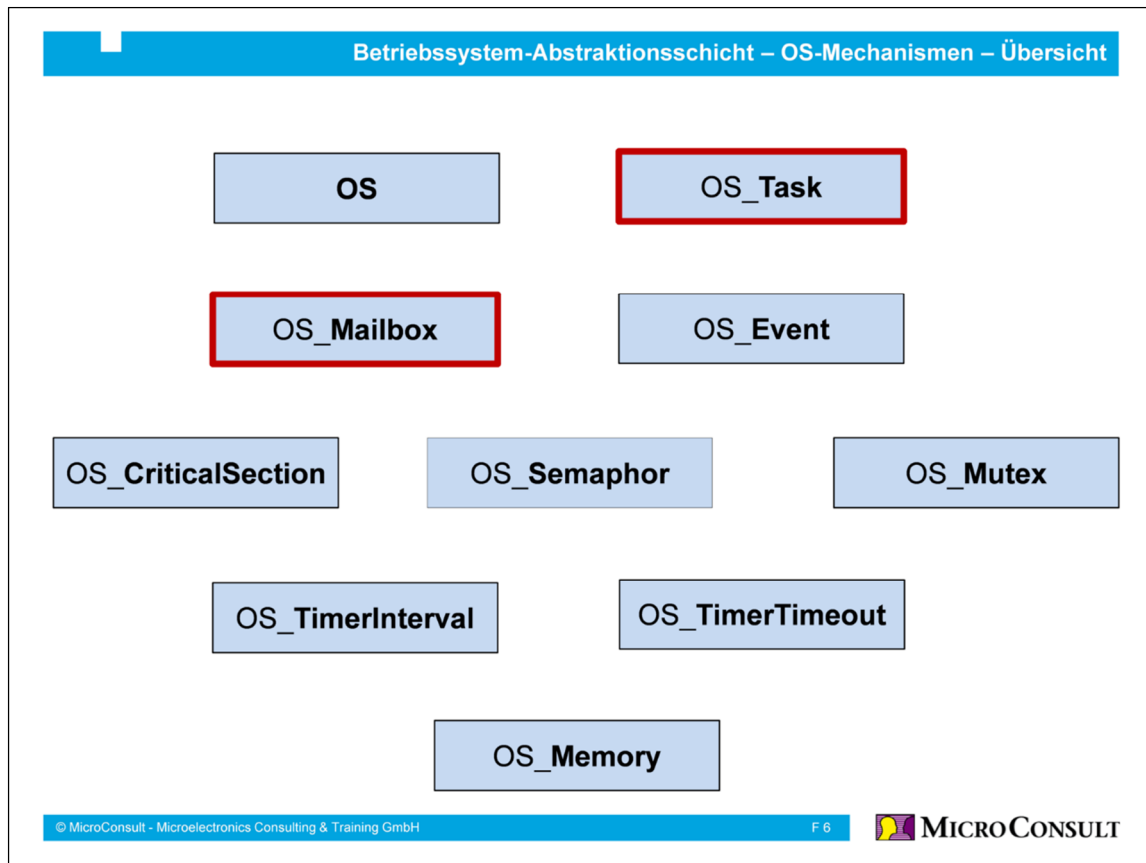
- Effizienz / Verbrauchsverhalten:
 - Verschlechterung der Laufzeit
 - Steigerung des Programmspeicher-Verbrauchs
- Abstraktion des kleinsten gemeinsamen Nenners potentieller Betriebssysteme
 - ➔ Einschränkung der Nutzung von Betriebsmitteln der Betriebssysteme

© MicroConsult - Microelectronics Consulting & Training GmbH
F 5

Der Hauptnutzen bei der Einführung einer Betriebssystem-Abstraktionsschicht liegt in der einfachen Austauschbarkeit des konkreten Betriebssystems, ohne dabei den Rest der Software adaptieren zu müssen.

3.2 OS-Mechanismen

3.2.1 Übersicht



Jedes Betriebssystem bietet vielen Mechanismen. Die hier dargestellten sind in nahezu jedem enthalten und bieten somit eine gute Grundlage für die Abstraktion.

OS

Enthält Typdefinitionen und die Möglichkeit, das Betriebssystem zu booten.

OS_Task

Der Scheduler des Betriebssystems führt die Tasks nach einem Algorithmus (z.B. prioritäts- und / oder zeitgesteuert) aus.

OS_Mailbox

Die Mailbox ist ein Konzept, um zwischen Tasks und Tasks bzw. auch zwischen ISR und Tasks Informationen zu kommunizieren.

OS_Event

Bei parallelen Abläufen kann es wichtig sein, dass sich Tasks untereinander und mit ISRs an definierten Punkten synchronisieren. Dies kann durch ein Event erfolgen. Ein Event ist damit eine 1-Bit-Kommunikation.

OS_CriticalSection

Um kritische Ausführungssequenzen in einer Task vor der Unterbrechung durch den Scheduler bzw. andere Tasks zu schützen, kommt die Critical-Sektion zum Einsatz.

OS_Semaphore

Um den gleichzeitigen gemeinsamen Zugriff auf eine Ressource schützen, kommt die Semaphore (binär und oder zählend) Critical-Sektion zum Einsatz.

OS_Mutex

Der Mutex ist prinzipiell mit der binären Semaphore vergleichbar, unterscheidet sich aber in den detaillierten Algorithmen.

OS_TimerInterval

Intervall-Timer erlauben es, Tasks oder Tasksequenzen periodisch zu wiederholen.

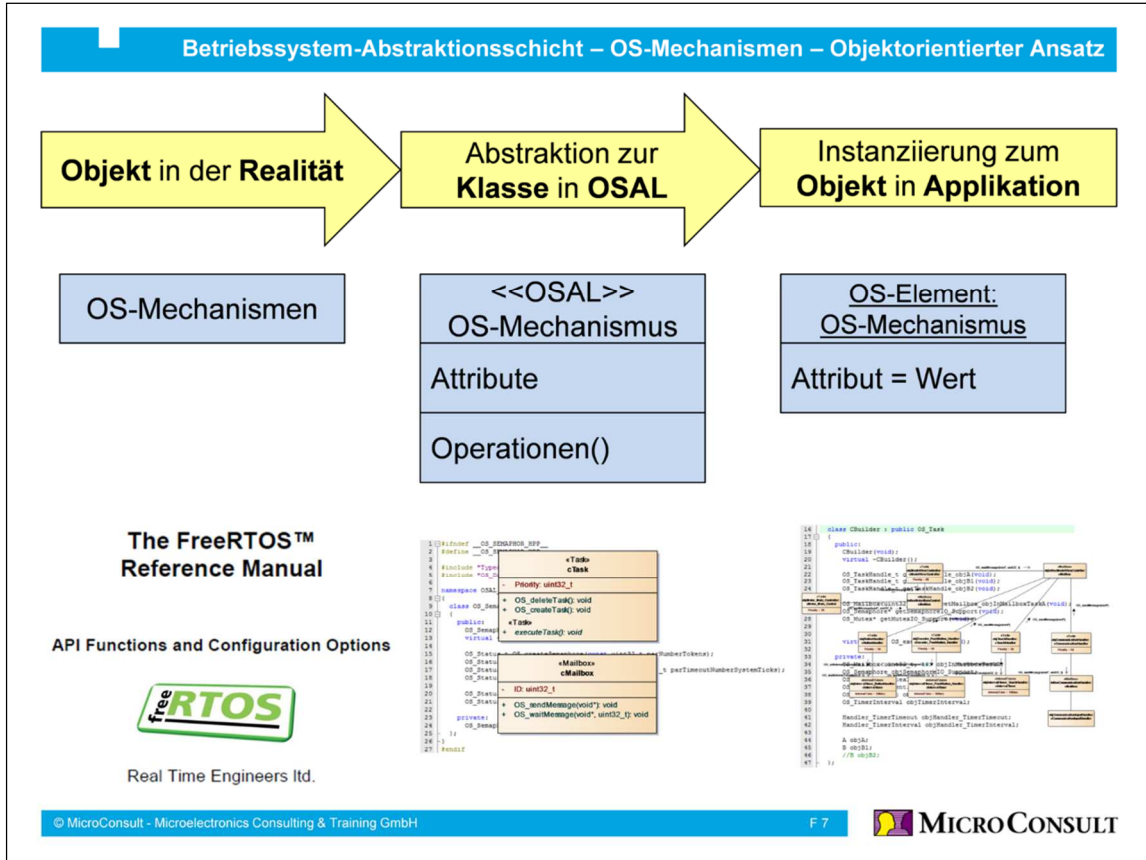
OS_TimerTimeout

Zeitüberwachungen in Tasks sind mit einem Timeout-Timer möglich.

OS_Memory

Hier enthalten sind Algorithmen zum sicheren dynamischen Speichermanagement, klassischerweise Speicherpool- / Speicherblock-Konzepte.

3.2.2 Objektorientierter Ansatz



Wie bei der Programmierung von Hardware-Treibern bietet sich auch für die Umsetzung der Betriebssystem-Abstraktion der objektorientierte Ansatz mit Klassen und Objekten an. Die Objekte in der Realität sind im Betriebssystem-Manual zu finden. Dort sind die API-Calls in logische Gruppen unterteilt. Diese Gruppen, z.B. Task-Management, Synchronisation, Kommunikation usw., repräsentieren die Objekte in der Realität. Diese müssen nun sinnvoll zu Klassen im OSAL abstrahiert bzw. gruppiert werden, z.B. `OS_Mailbox`. Dabei erfolgt die Ausstattung mit Attributen und Operationen. Die OSAL-Klassen werden in der Applikation zu konkreten Softwareobjekten instanziiert, z.B. `objInMailboxTaskControl`.

4 Task-Abstraktion

4.1 FreeRTOS API Übersicht

Task-Abstraktion – FreeRTOS API-Übersicht

OS_Task

Task Creation

- [TaskHandle_t \(type\)](#)
- [xTaskCreate\(\)](#)
- [vTaskDelete\(\)](#)

Task Control

- [vTaskDelay\(\)](#)
- [vTaskDelayUntil\(\)](#)
- [uxTaskPriorityGet\(\)](#)
- [vTaskPrioritySet\(\)](#)
- [vTaskSuspend\(\)](#)
- [vTaskResume\(\)](#)
- [xTaskResumeFromISR\(\)](#)

Task Utilities

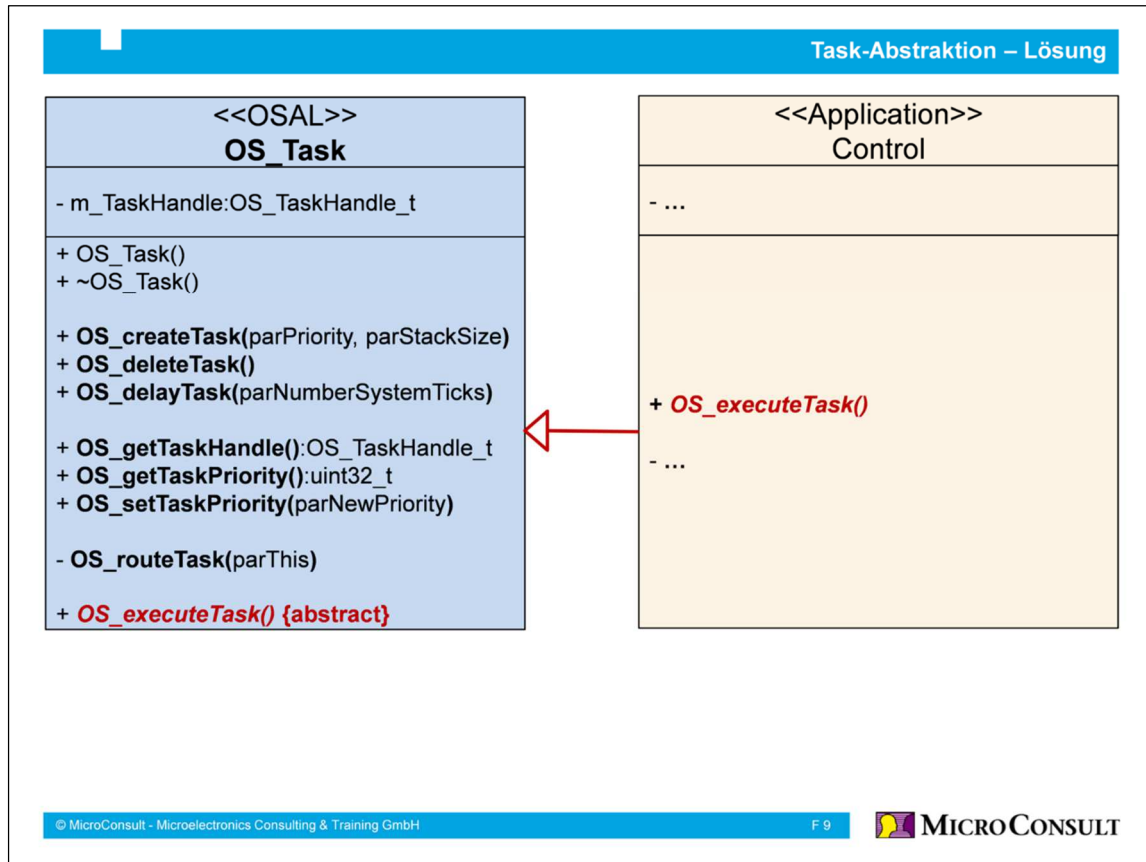
- [uxTaskGetSystemState\(\)](#)
- [xTaskGetApplicationTaskTag\(\)](#)
- [xTaskGetCurrentTaskHandle\(\)](#)
- [xTaskGetIdleTaskHandle\(\)](#)
- [uxTaskGetStackHighWaterMark\(\)](#)
- [eTaskGetState\(\)](#)
- [pcTaskGetTaskName\(\)](#)
- [xTaskGetTickCount\(\)](#)
- [xTaskGetTickCountFromISR\(\)](#)
- [xTaskGetSchedulerState\(\)](#)
- [uxTaskGetNumberOfTasks\(\)](#)
- [vTaskList\(\)](#)
- [vTaskStartTrace\(\)](#)
- [ulTaskEndTrace\(\)](#)
- [vTaskGetRunTimeStats\(\)](#)
- [vTaskSetApplicationTaskTag\(\)](#)
- [xTaskCallApplicationTaskHook\(\)](#)



© MicroConsult - Microelectronics Consulting & Training GmbH
F 8

 **MICROCONSULT**

4.2 Lösung

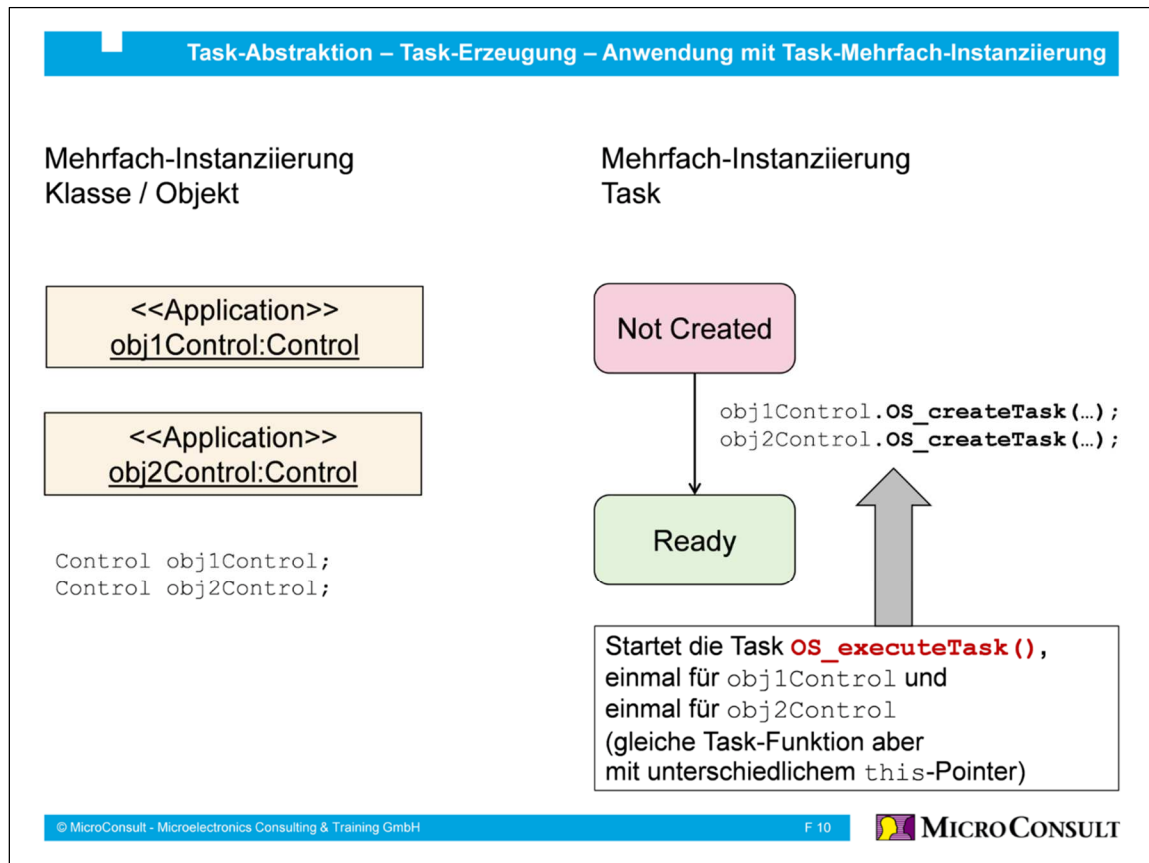


Die Klasse OS_Task enthält als Attribut das Task-Handle m_TaskHandle mit allen Task-relevanten Daten. Um mit der Task in der Applikation zu arbeiten, gibt es die zentralen Steuerungsoperationen (OS_createTask(), OS_delete_Task(), OS_delayTask()) und zusätzlich einige Hilfsoperationen (OS_getTaskHandle(), OS_getTaskPriority(), OS_setTaskPriority()) im public Bereich.

Zusätzlich ist die rein virtuelle Operation OS_executeTask() als eine Taskfunktion deklariert. Dies ermöglicht das Konzept, dass jede Applikationsklasse, die eine Taskfunktion enthalten soll, von der Basisklasse OS_Task erben muss.

4.3 Task-Erzeugung

4.3.1 Anwendung mit Task-Mehrfach-Instanziierung



Betriebssysteme erlauben es, eine Taskfunktion mehrfach zu unterschiedlichen Tasks zur Laufzeit zu starten. Das bedeutet, dass eine Applikationsklasse `Control`, die eine Taskfunktion enthält, mehrfach instanzierbar sein muss. Für beide Objekte der Applikationsklasse `Control` muss ein `OS_createTask()` aufrufbar sein, mit der Konsequenz, dass danach der Scheduler zwei Tasks (für jedes Objekt eine) der einen Taskfunktion ausführt. Das mit der Operation `OS_createTask()` gebundene Objekt bindet der Compiler mit dem `this`-Pointer statisch. Die Übergabe des `this`-Pointers wird im folgenden Abschnitt erläutert.

4.3.2 Manuelle Übergabe des this-Pointers

Task-Abstraktion – Task-Erzeugung – Manuelle Übergabe des this-Pointers

```

OS_Status_t OS_Task::OS_createTask(uint32_t parPriority, uint16_t parStackSize)
{
    ...
    loc_Status = xTaskCreate(
        OS_Task::OS_routeTask, // task function
        "Task-Name",           // task name for debug
        parStackSize,          // task stack size
        reinterpret_cast<void*>(this), // task function parameter
        parPriority,            // task priority
        &m_TaskHandle);        // task handle
    ...
}
            
```

<<OSAL>>

OS_Task
 - m_TaskHandle: OS_TaskHandle_t
 + OS_Task()
 + ~OS_Task()
 + OS_createTask(parPriority, parStackSize)
 + OS_deleteTask()
 + OS_delayTask(parNumberSystemTicks)
 + OS_getTaskHandle(): OS_TaskHandle_t
 + OS_getTaskPriority(): uint32_t
 + OS_setTaskPriority(parNewPriority)
 - OS_routeTask(parThis)
 + OS_executeTask() {abstract}

```

void OS_Task::OS_routeTask(void* parThis)
{
    reinterpret_cast<OS_Task*>(parThis)->OS_executeTask();
}
            
```

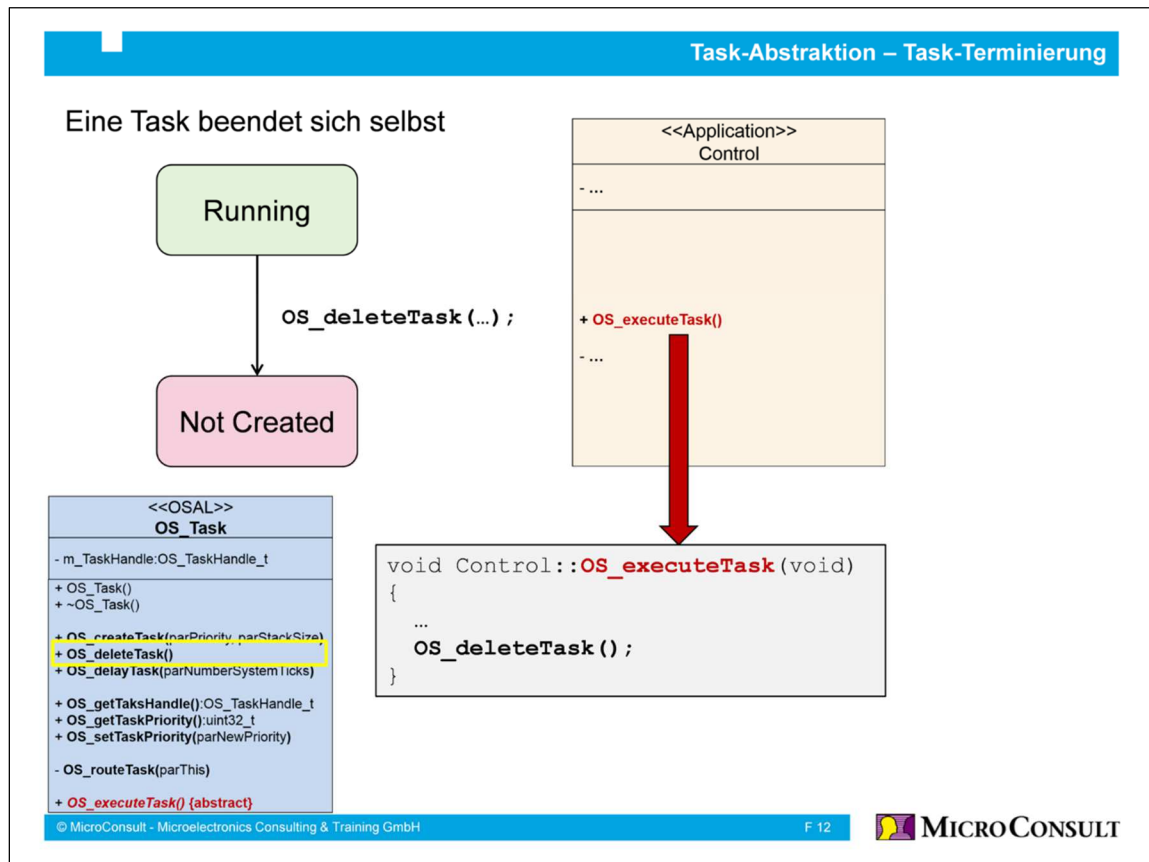
© MicroConsult - Microelectronics Consulting & Training GmbH
F 11

In der Operation `OS_createTask()` erfolgt der Aufruf der FreeRTOS Funktion `xTaskCreate()`. Diese kennt jedoch keinen `this`-Pointer für eine Taskfunktion. Die eigentliche Taskfunktion wird nun in der Klasse `OS_Task` durch die Hilfsoperation `OS_routeTask()` ersetzt. Diese erhält den `this`-Pointer als Parameter. Mit diesem Parameter ruft die Operation `OS_routeTask()` die eigentliche Taskfunktion `OS_executeTask()` auf.

Die FreeRTOS Funktion `xTaskCreate()` bekommt als Taskfunktion `OS_routeTask()` übergeben und als Taskfunktionsparameter den `this`-Pointer.

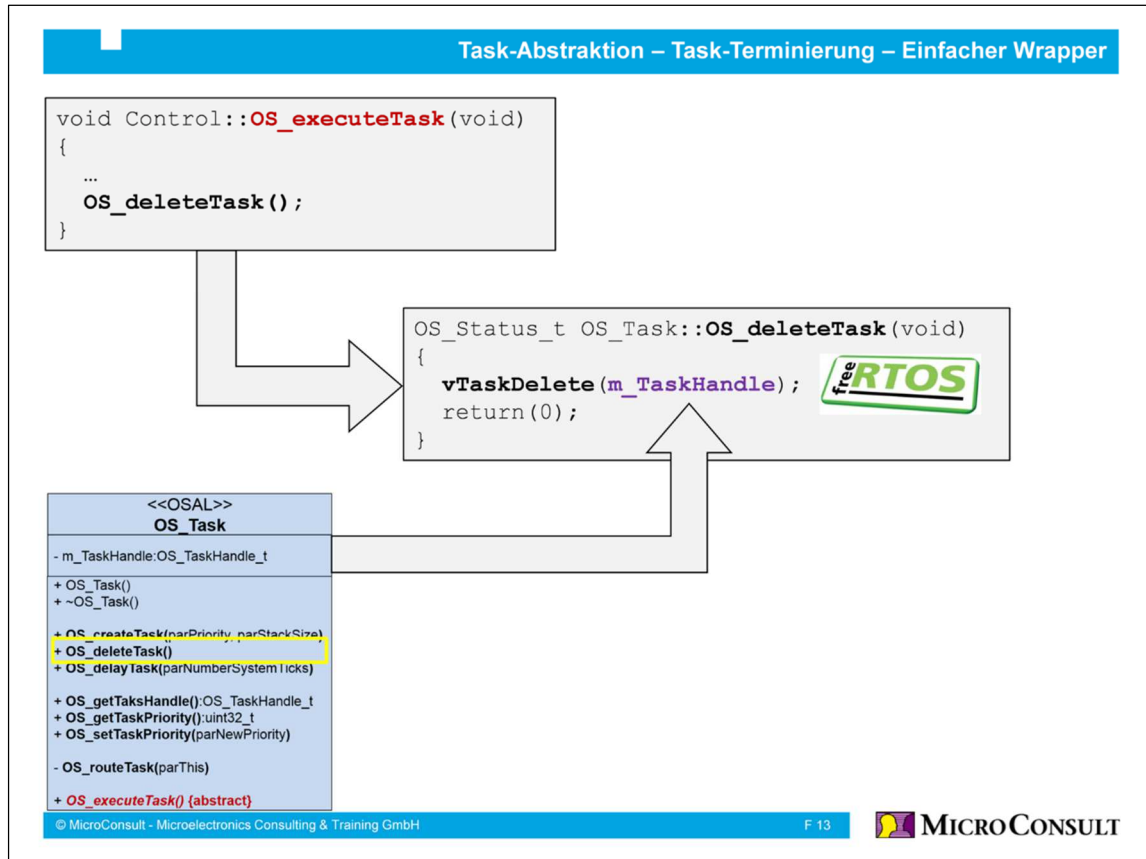
Somit ist die Mehrfach-Instanziierung von Tasks möglich.

4.4 Task-Terminierung



Die Klasse `OS_Task` enthält die Operation `OS_deleteTask()`. Durch deren Aufruf kann sich eine Task selbst beenden. Die Taskfunktion `OS_executeTask()` kann diese am Ende aufrufen, um sich selbst als Task zu beenden.

4.4.1 Einfacher Wrapper



Die Operation `OS_deleteTask()` ist ein klassischer Wrapper. Sie enthält die Umsetzung auf die von FreeRTOS bereitgestellte Funktion `vTaskDelete()`. Um sicherzustellen, dass diese Funktion die richtige Task beendet, erhält sie als Parameter den entsprechenden Task-Handle der Klasse `OS_Task`.

5 Mailbox-Abstraktion

5.1 FreeRTOS API Übersicht

Mailbox-Abstraktion – FreeRTOS API-Übersicht

OS_Mailbox

☰
Queues

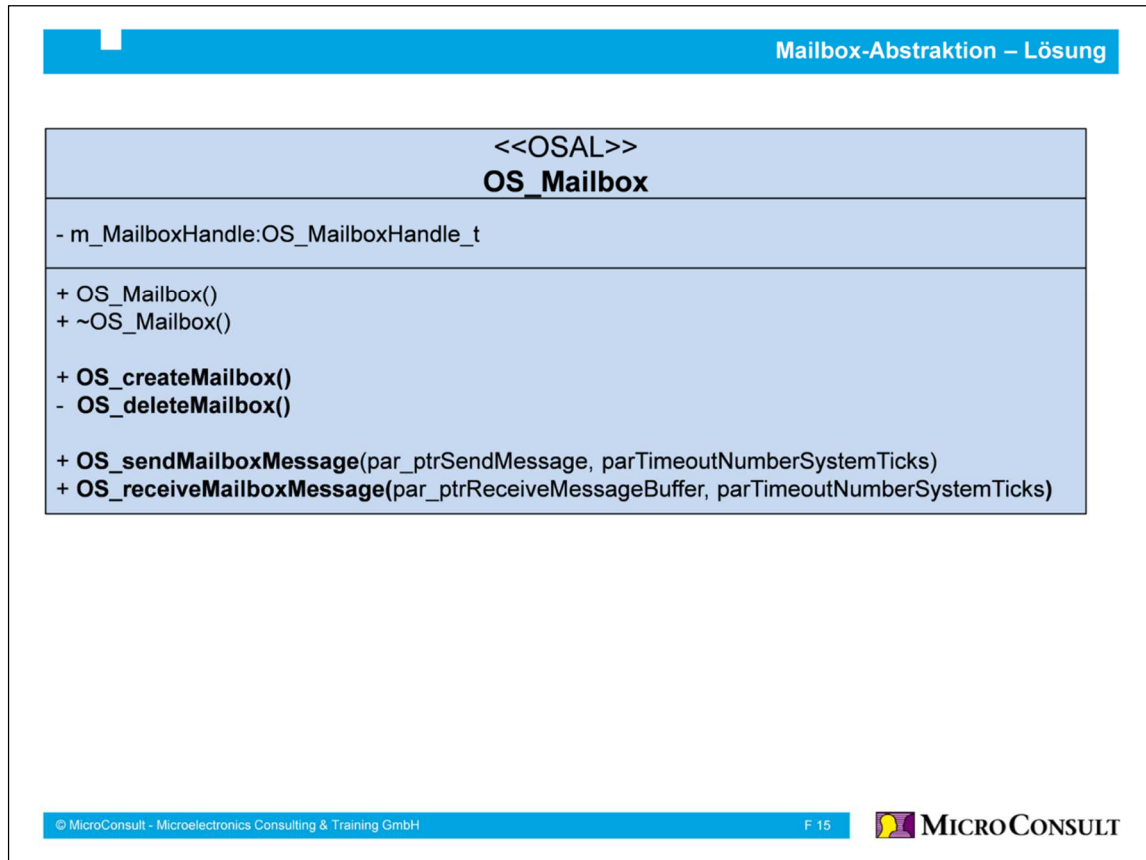
[uxQueueMessagesWaiting\(\)](#)
[uxQueueMessagesWaitingFromISR\(\)](#)
[uxQueueSpacesAvailable\(\)](#)
[xQueueCreate\(\)](#)
[vQueueDelete\(\)](#)
[xQueueReset\(\)](#)
[xQueueSend\(\)](#)
[xQueueSendToBack\(\)](#)
[xQueueSendToFront\(\)](#)
[xQueueReceive\(\)](#)
[xQueueOverwrite\(\)](#)
[xQueueOverwriteFromISR\(\)](#)
[xQueuePeek\(\)](#)
[xQueuePeekFromISR\(\)](#)
[xQueueSendFromISR\(\)](#)
[xQueueSendToBackFromISR\(\)](#)
[xQueueSendToFrontFromISR\(\)](#)
[xQueueReceiveFromISR\(\)](#)
[vQueueAddToRegistry\(\)](#)
[vQueueUnregisterQueue\(\)](#)
[xQueueIsQueueFullFromISR\(\)](#)
[xQueueIsQueueEmptyFromISR\(\)](#)



© MicroConsult - Microelectronics Consulting & Training GmbH
F 14


MICROCONSULT

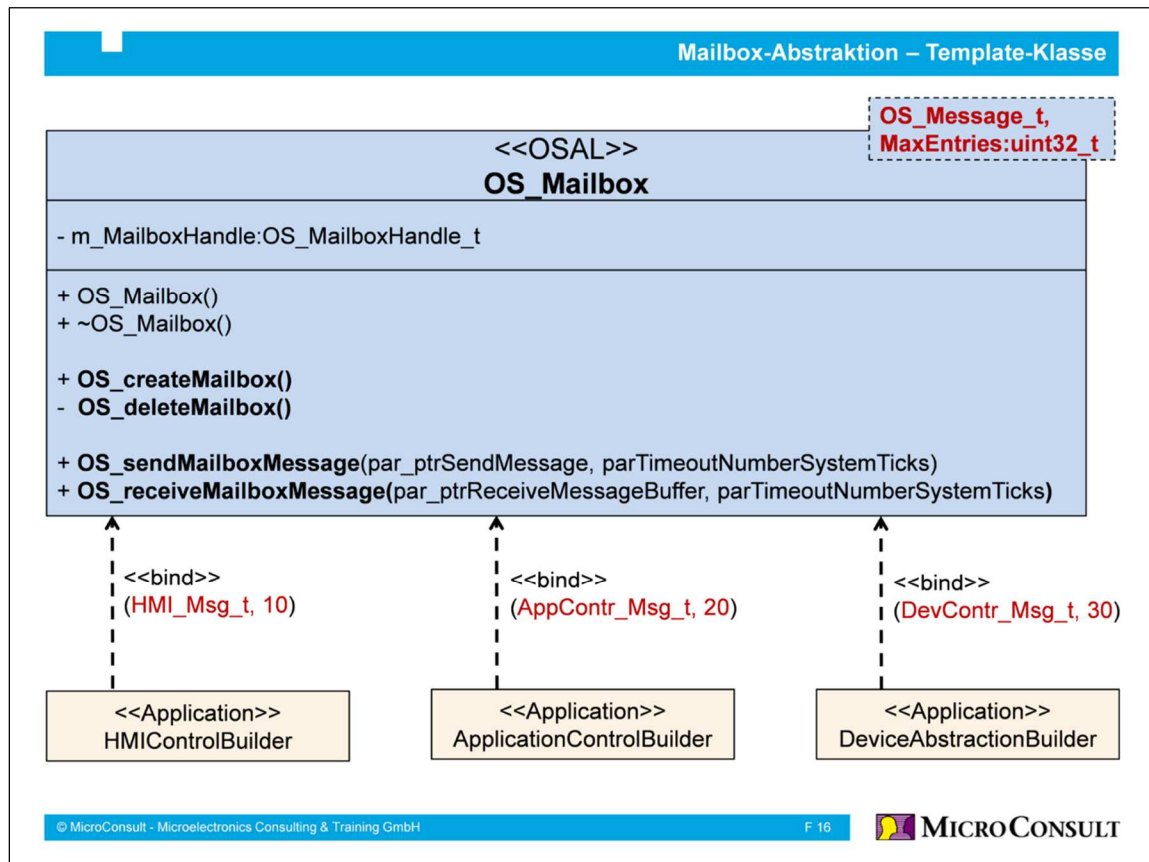
5.2 Lösung



Die Klasse OS_Mailbox enthält als Attribut das Mailbox-Handle m_MailboxHandle mit allen Mailbox-relevanten Daten. Um mit der Mailbox in der Applikation zu arbeiten, gibt es die zentralen Operationen (OS_createMailbox(), OS_deleteMailbox(), OS_sendMailboxMessage() und OS_receiveMailboxMessage()) im public Bereich.

Für die Mailbox entscheidende Parameter sind der **Nachrichtentyp** und die **maximale Anzahl von Nachrichten**, die in einer Mailbox gleichzeitig enthalten sein können. Die Klasse OS_Mailbox soll für unterschiedlichste Nachrichtentypen und Nachrichtenanzahlen nur einmal programmiert sein. Die beste Lösung dazu ist die Verwendung einer **Template-Klasse**.

5.3 Template-Klasse



Die Template-Klasse erhält als Parameter den Nachrichtentyp OS_Message_t und die maximale Anzahl von gleichzeitig aufnehmbaren Nachrichten MaxEntries. Die Template-Klasse ist somit mit unterschiedlichsten Template-Parametern in der Applikation anwendbar.

5.4 Mailbox-Erzeugung

5.4.1 Konstruktor versus eigenständige Operation

Mailbox-Abstraktion – Mailbox-Erzeugung – Konstruktor versus eigenständige Operation

```

template <typename OS_Message_t, uint32_t MaxEntries>
OS_Mailbox<OS_Message_t, MaxEntries>::OS_Mailbox(void)
{
    m_MailboxHandle = NULL;
    //OS_createMailbox();           // possible
}
            
```

```

template <typename OS_Message_t, uint32_t MaxEntries>
OS_Status_t OS_Mailbox<OS_Message_t, MaxEntries>::OS_createMailbox(void)
{
    m_MailboxHandle = xQueueCreate(           // allocates memory
                                MaxEntries,   // max. amount of messages
                                sizeof(OS_Message_t) // size of one message
                                );
    ...
}
            
```

© MicroConsult - Microelectronics Consulting & Training GmbH

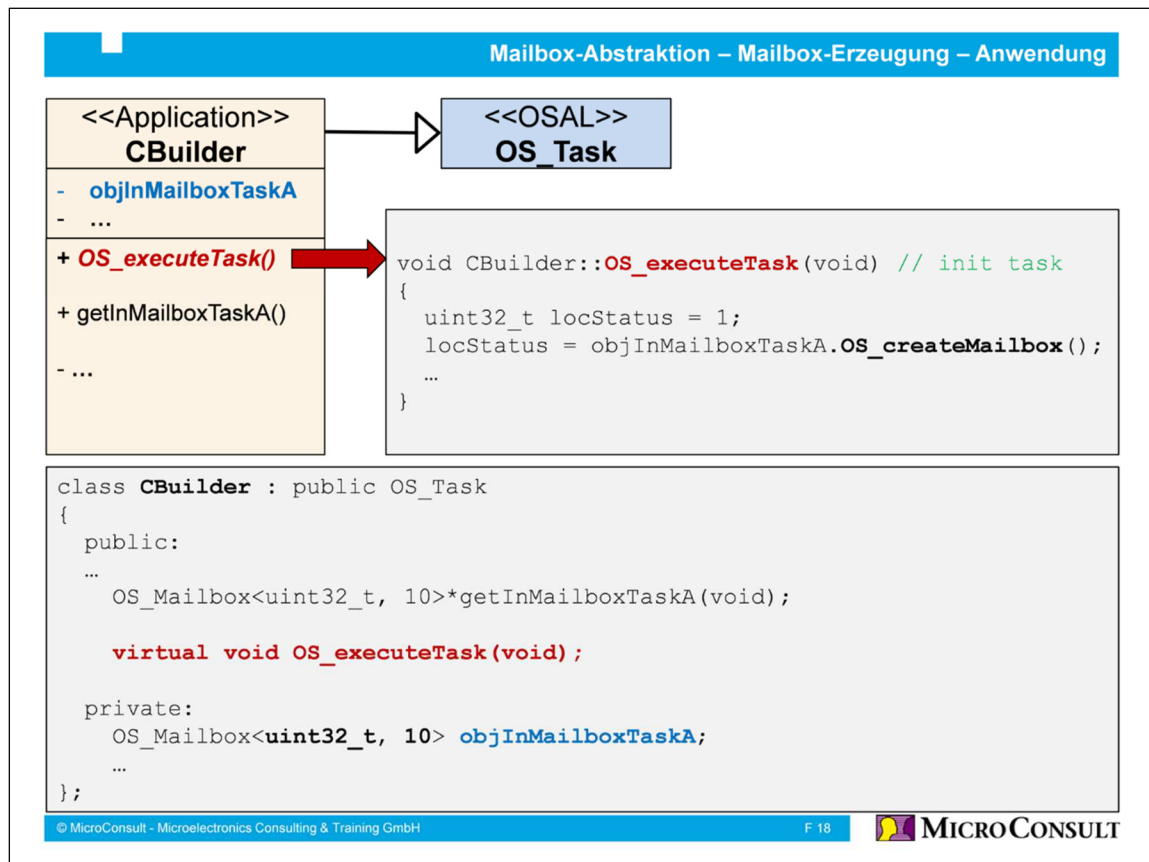
F 17

MICROCONSULT

Im Konstruktor der Klasse OS_Mailbox könnte direkt die Operation OS_createMailbox() aufgerufen werden. Da der Konstruktor aber keinen direkten Return-Wert zur Auswertung liefern kann, ist der getrennte Aufruf von Konstruktor und OS_createMailbox() empfehlenswert.

Die Template-Parameter OS_Message_t und MaxEntries lassen sich direkt in die FreeRTOS Funktion xQueueCreate() übergeben. Diese Funktion allokiert gleichzeitig zur Initialisierung auch den erforderlichen Speicher für die Mailbox.

5.4.2 Anwendung



Eine Erzeugungsklasse CBuilder könnte die benötigten Mailboxen als Attribute (eingebettete Objekte) über den Konstruktor erzeugen, z.B. objInMailboxTaskA.

Die Klasse CBuilder enthält die Initialisierungstask OS_executeTask(), die u.a. die Operation OS_createMailbox() aufruft. Um anderen Tasks die Möglichkeit zu bieten, mit der Mailbox zu interagieren, enthält die Klasse CBuilder zusätzlich die Hilfsoperation getInMailboxTaskA(), die das für den Zugriff erforderliche Mailbox-Handle zurückgibt.

6 Weitere Abstraktionskonzepte mit C++

6.1 Diskussionspunkte

Weitere Abstraktionskonzepte mit C++ – Diskussionspunkte

- `#define` für einfache Typumbenennung in einem zentralen Header-File, z.B. `OS.h`
- Verwendung der Konstruktoren und Destruktoren für die automatischen Aufrufe `OS_create...()` und `OS_delete...()`
- Einführung von Interfaceklassen für jeden OS-Mechanismus
- `static` Elemente für die Verwendung im Interrupt-Kontext
- Ausnutzung von Default-Parametern bei Operationen, z.B.
`...OS_createTask(..., uint16_t parStackSize = 100);`
- Ausnutzung von RTTI für automatische Vergabe des Task-Namens
`...OS_createTask(..., "Task-Name", ...);`

6.2 Quellen standardisierter Interfaces für eine Betriebsabstraktion

Quellen standardisierter Interfaces für eine Betriebsabstraktion

- OSEK (**O**ffene **S**ysteme und deren Schnittstellen in der **E**lektronik im **K**fz)
<http://www.osek-vdx.org/>
- AUTOSAR (**AUTO**motive **O**pen **S**ystem **AR**chitecture)
<http://www.autosar.org/>
- CMSIS (**C**ortex® **M**icrocontroller **S**oftware Interface **S**tandard)
<http://www.arm.com>
- POSIX (**P**ortable **O**perating **S**ystem Interface)
<http://standards.ieee.org/findstds/standard/14515-2-2003.html>
- NASA OSAL (**N**ational **A**eronautics and **S**pace **A**dministration)
<https://github.com/nasa/osal>

© MicroConsult - Microelectronics Consulting & Training GmbH

F 20

 **MICROCONSULT**

7 Resümee

Resümee

Der **objektorientierte Ansatz** eignet sich für
die Betriebssystem-Abstraktion perfekt

C++ unterstützt
den objektorientierten Ansatz sehr gut

C++ unterstützt
mit anderen Programmierkonstrukten die Betriebssystem-Abstraktion sehr gut

© MicroConsult - Microelectronics Consulting & Training GmbH

10.11.2014

F 21

 **MICROCONSULT**