

LUCA Protocol Factory

1. Tutorial

1.1. Einführung

1.1.1. Was ist die Protocol Factory?

1.1.2. Was gehört zur Protocol Factory?

1.1.3. Überblick über die Funktionsweise der Protocol Factory

1.2. Datenstrukturen

1.2.1. Portabilität

1.2.2. Datenblock

1.2.3. Messagetypen

1.2.4. Message

1.2.5. Modulinformationen

1.2.6. Queues

1.3. Ein einfaches Beispielmodul

1.3.1. Taten mit Daten

1.3.2. Die Exporttabelle

1.3.3. Die open-Funktion

1.3.4. Die close-Funktion

1.3.5. Die wput-Funktion

1.3.6. Die rput-Funktion

1.3.7. Umwandeln der Daten

1.3.8. Einfache Parameter

1.3.9. Kompilieren und Linken des Moduls

1.3.10. Test- und Tracemöglichkeiten

1.4. XONXOFF: Ein Beispielprotokollmodul mit Flußkontrolle

1.4.1. Die Serviceroutinen

1.4.2. Eine einfache Serviceroutine

1.4.3. Ein- und Ausschalten der Queue

1.4.4. Senden der Steuerinformation

1.4.5. Füllstände und -grenzen

1.5. Protokollverhandlung

1.5.1. Einführung: Protokollverhandlung

1.5.2. Verbindungsaufbau im Protokollmodul

1.5.3. Daten von der Gegenstelle abprüfen

1.5.4. Daten an die Gegenstelle senden

1.6. Treibermodule

1.6.1. Einführung: Treibermodule

1.6.2. Verbindungsaufbau im Treibermodul

1.6.3. Daten an die Anwendung senden

1.6.4. Die Leseserviceroutine des Treibers

1.6.5. Verbindungsabbau

1.7. Spezialfunktionen

1.7.1. Protokolle unter sich

1.7.2. Automatisches Laden von Treibern

1.7.3. Schützen von Programmabschnitten

1.7.4. Der Protokollevent

1.1. Einführung

1.1.1. Was ist die Protocol Factory?

1.1.2. Was gehört zur Protocol Factory?

1.1.3. Überblick über die Funktionsweise der Protocol Factory

1.1.3.1. Bestandteile der LUCA.DLL

1.1.3.2. Einen Port öffnen und schließen

1.1.3.3. Parameterübergabe

1.1.3.4. Protokollmodule

1.1.3.5. Treibermodule

1.1.3.6. Messages

1.1.3.7. Queues

1.1.3.8. Ereignisse

1.1.3.9. Zeitsteuerung

1.1.1. Was ist die Protocol Factory?

Diese Dokumentation beschreibt die LUCA Protocol Factory.

Mit der Protocol Factory ist es möglich, eigene Protokolle und Treiber zu entwickeln, die völlig transparent in die LUCA-Programmierung eingegliedert werden können.

Wichtig: Wenn Sie keine eigenen Protokolle und Treiber implementieren möchten, brauchen Sie die Protocol Factory nicht zu benutzen. Sie brauchen dann auch dieses Handbuchkapitel nicht zu lesen.

Wenn Sie mit der Protocol Factory eigene Protokoll- und Treibermodule erstellen wollen, müssen Sie mit der Programmiersprache C arbeiten. Die fertigen Module können allerdings auch in anderen Entwicklungsumgebungen (z.B. Visual Basic) eingesetzt werden.

Diese Dokumentation ist aufgeteilt in eine Einleitung, eine Beschreibung der wichtigsten Datenstrukturen, mehrere Kapitel mit Beispielanwendungen und einer alphabetischen Referenz aller Strukturen und Funktionen.

Falls Sie zu den Schnellstartern gehören und lieber sofort loslegen möchten, sollten Sie sich zuerst die einfacheren Beispielmuster ansehen und das dazugehörige Kapitel lesen. Das erste Beispielmuster beschäftigt sich mit reiner Datenveränderung, das zweite mit Verbindungsaufbau, das dritte mit Flußkontrolle und das letzte mit Treiberprogrammierung.

1.1.2. Was gehört zur Protocol Factory?

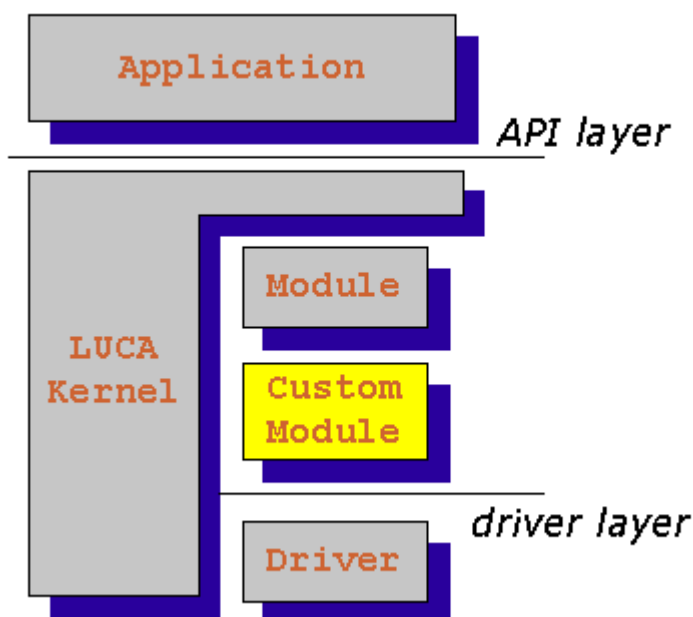
Die Protocol Factory besteht hauptsächlich aus dieser Dokumentation, aus einer C-Source-Datei, die den Code für die Kommunikation mit der LUCA.DLL enthält, und aus einer C-Headerdatei. Desweiteren gibt es einige Beispielmuster, die zeigen, wie häufig vorkommende Kommunikationsprobleme einfach gelöst werden können.

1.1.3. Überblick über die Funktionsweise der Protocol Factory

- 1.1.3.1. Bestandteile der LUCA.DLL
- 1.1.3.2. Einen Port öffnen und schließen
- 1.1.3.3. Parameterübergabe
- 1.1.3.4. Protokollmodule
- 1.1.3.5. Treibermodule
- 1.1.3.6. Messages
- 1.1.3.7. Queues
- 1.1.3.8. Ereignisse
- 1.1.3.9. Zeitsteuerung

1.1.3.1. Bestandteile der LUCA.DLL

LUCA besteht im wesentlichen aus einem Kern und aus mehreren Protokoll- und Treibermodulen.



Alle Module sind gleichberechtigt und tauschen untereinander Messages aus, was man sich so ähnlich vorstellen kann wie die Messages, die grafische Benutzeroberflächen wie Windows verwenden. Die Messages, die zwischen den Modulen ausgetauscht werden, teilen sich auf in Daten- und Kontrollinformationen. Kontrollinformationen können wiederum medienunabhängig oder protokollspezifisch sein.

Kein Modul darf die Rechnerkapazität übermäßig lange in Anspruch nehmen und muß daher ereignisgesteuert implementiert werden. Ein Scheduler innerhalb vom LUCA-Kern sorgt dafür, daß bei entsprechendem Datenaufkommen die unterschiedlichen Module, die an einer Verbindung beteiligt sind, nacheinander aufgerufen werden.

1.1.3.2. Einen Port öffnen und schließen

Jedes LUCA-Modul entspricht im Link Identifier von Vopen() genau dem Namen, der dort angegeben ist, d.h. es gibt ein Modul ASYNC, COM, TCP, SOCKET usw. Kein Modul muß sich selbst um den Link Identifier beim Öffnen kümmern, die Interpretation des Link Identifiers ist Aufgabe des LUCA-Kerns. Das Laden und Öffnen eines LUCA-Ports kann man sich so vorstellen:

Interpretation des Link Identifiers

Der Link Identifier wird zunächst in Worte zerschnitten ("geparst"). Aus TEXT,CR=15/ASYNC/COM:2 wird also

```
COM
  SUBADDR 2
ASYNC
TEXT
  CR 15
```

Laden einzelner Module

Ist die Schreibweise korrekt, fängt der LUCA-Kern an, das unterste Modul (im Beispiel COM), das sich Treibermodul nennt, zu öffnen bzw. zu laden. Ist dies ohne Probleme möglich, wird das zweite Modul (im Beispiel ASYNC) geladen. Tritt ein Fehler auf, wird das Treibermodul in diesem Fall automatisch geschlossen.

Sind für ein Modul Parameter angegeben, werden diese vom LUCA-Kern an das Modul weitergegeben. Dazu später mehr.

Verbindungsaufbau

Nachdem alle Module geladen und initialisiert und die Parameter korrekt verhandelt worden sind, wird der Verbindungsaufbau initiiert.

Dazu wird nacheinander vom Treiber her jede "Protokollschicht" bis zum vollständigen Aufbau durchlaufen. In der Praxis geht das natürlich sehr schnell, da keine Daten übertragen werden müssen.

Datenübertragung

Für die eigentliche Datenübertragung erhält jedes Modul Datenpakete sowohl von "oben", von der Anwendung, als auch von "unten", vom Treiber.

Die Übertragung vollzieht sich also immer bidirektional, wobei u.U. in eine Richtung Daten nur transparent weitergereicht oder verworfen werden.

Verbindungsabbau

Die Verbindung wird in umgekehrter Richtung, in der sie aufgebaut wurde, wieder abgebaut. Das oberste Protokollmodul baut daraufhin *seine* entsprechende Protokollschicht ab, woraufhin das darunterliegende Modul *sein* Protokoll abwickelt usw. bis das Treibermodul die Verbindung kappt.

Freigeben der Module

Nachdem ein Port nicht mehr benutzt wird, fängt der LUCA-Kern an, die einzelnen Module in genau der umgekehrten Reihenfolge wie beim Laden freizugeben. Jedes Modul erhält damit die Möglichkeit, Speicher freizugeben und Timer zu löschen.

1.1.3.3. Parameterübergabe

Genauso wie die Steuerung des Verbindungsauf- und -abbaus durch Messages erfolgt, werden auch Parameter an Module in Form von Messages weitergegeben und von Modulen als Messages ausgelesen. Die Struktur der Messages ist für den Programmierer unerheblich, da für die Parameterbehandlung der Protocol Factory-Entwickler vorgefertigte LUCA-Funktionen zur Verfügung stehen.

Damit die Implementierung eines Moduls weiter vereinfacht wird, erhält das Modul die Adresse (der Teil, der hinter einem Doppelpunkt im Link Identifier steht) durch den Parameter SUBADDR. Außerdem wird nicht unterschieden zwischen den Parametern, die im Link Identifier hinter dem Modul stehen und den Parametern, die durch **Vsetpar** gesetzt werden können. Für die Ausnahmefälle, bei denen Parameter nach einem Verbindungsaufbau nicht mehr setzbar sind, ist das Modul selbst verantwortlich. Das gleiche gilt für die korrekte Behandlung von schreibgeschützten Parametern.

1.1.3.4. Protokollmodule

Ein Protokollmodul ist ein Modul, das Daten in beide Richtungen an andere Module weitergibt. Es gibt in LUCA keine prinzipiellen Einschränkungen hinsichtlich der Anordnung der Protokollmodule untereinander. Nur in der Dokumentation wurde indessen Wert auf leichte Verständlichkeit gelegt und deshalb darauf hingewiesen, in welcher Kombination die Module in der Praxis miteinander funktionieren.

Beispiele für Protokollmodule sind ATMODEM, ZMODEM, MIME, FTP. Viele Entwickler werden sicherlich zuerst Protokollmodule implementieren, um die vorhandenen LUCA-Treiber für spezielle Zwecke zu benutzen.

1.1.3.5. Treibermodule

Ein Treibermodul hat keine weiteren Module in der Protokollhierarchie unter sich. Anders ausgedrückt: es steht im Link Identifier ganz rechts.

Treibermodule sind meist immer plattformabhängig und sprechen teilweise direkt die Hardware des Rechners an. Beispiele für Treibermodule sind COM, SOCKET, CAPI und ECHO.

1.1.3.6. Messages

Als Message bezeichnen wir eine Datenstruktur, die es Modulen ermöglicht, untereinander Daten und Kontrollinformationen auszutauschen.

Diese Datenstruktur spielt eine zentrale Rolle in der Protocol Factory.

Im Gegensatz zu den Messages, die man z.B. im Windows API benutzt, haben LUCA-Messages eine variable Größe. Man kann theoretisch mehrere Kilobytes Daten in einer Message unterbringen. Außerdem haben LUCA-Messages eine feste äußere Schale, die nicht vom Typ der Message abhängt. Für alle, die es gewohnt sind mit Objekthierarchien zu arbeiten, kann man sich eine LUCA-Message als ein Basisobjekt vorstellen, das einige Methoden bereitstellt, wie z.B. das Weiterleiten an das nächste Modul, Daten speichern usw.

1.1.3.7. Queues

Eine Queue ist eine Warteschlange, die Daten oder Informationen in Form von Messages für ein Modul bereithält.

Zu jedem Modul gehört eine Warteschlange in Senderichtung und in Empfangsrichtung, die man sich als First In / First Out (FIFO) Speicher vorstellen kann. Mit bestimmten Routinen lassen sich Messages in die jeweiligen Queues schreiben, wobei sich die Funktionen für Empfangs- und Senderichtung nicht unterscheiden.

1.1.3.8. Ereignisse

Treiber- und Protokollmodule können verschiedene Ereignisse in der API-Schicht auslösen. Diese Ereignisse kann man unterteilen in

- Verbindungsaufbau und Verbindungsabbau (V_CONNECT, V_DISCONNECT)

Diese Ereignisse treten erst ein, wenn das oberste Protokollmodul im Protokollstack an die Anwendung einen erfolgreichen Verbindungsauf- oder -abbau meldet.

- Datenaustausch (V_CANREAD, V_CANWRITE)

Diese Ereignisse hängen von der Anzahl der Datenblöcke ab, die die Anwendung bereits gelesen oder geschrieben hat.

- Fehler (V_ERR)

Ein Fehlerereignis führt zum Abbruch der aktuellen API-Funktion und beendet die Verbindung.

- Sonstige (V_NONSTD, V_PROTOCOL)

NONSTD- und PROTOCOL-Ereignisse werden vom LUCA-Kern zwischengespeichert und können von der Anwendungsseite durch Vgetpar()-Aufrufe abgefragt werden.

1.1.3.9. Zeitsteuerung

Der LUCA-Kern stellt den Modulen systemunabhängige Zeitgeberfunktionen zur Verfügung. Mit diesen Zeitgebern können bestimmte Protokollabläufe zeitlich kontrolliert werden.

1.2. Datenstrukturen

1.2.1. Portabilität

1.2.2. Datenblock

1.2.3. Messagetypen

1.2.4. Message

1.2.4.1. Messageblock

1.2.4.2. Messageblöcke mit mehreren Teilen

1.2.4.3. Allokieren

1.2.4.4. Daten speichern

1.2.4.5. Daten auslesen

1.2.4.6. Was passiert, wenn die Anwendung Vread() aufruft?

1.2.4.7. Freigeben oder Weitergeben von Messages

1.2.5. Modulinformationen

1.2.6. Queues

1.2.6.1. Übersicht: Queues

1.2.6.2. Füllstand

1.2.6.3. Daten ablegen

1.2.6.4. Daten auslesen

1.2.6.5. Serviceroutinen

1.2.1. Portabilität

Die Protocol Factory erleichtert die Portierung durch vordefinierte Datentypen und Datenstrukturen, so daß nur geringe bis keine Anpassungen an andere Betriebssysteme für Protokollmodule notwendig werden.

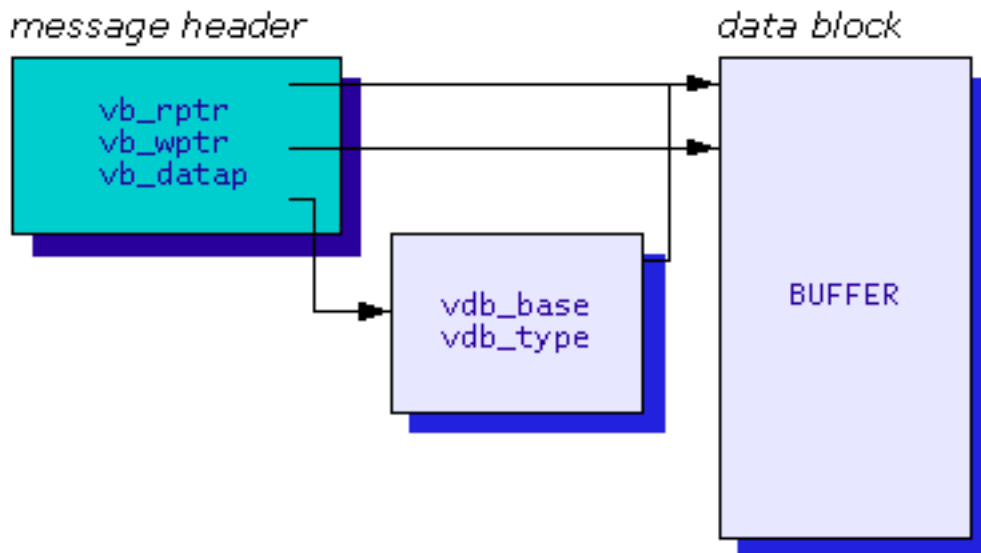
Folgende Typen sind für allgemeine Zwecke in `lucaPf.h` definiert:

<code>vbyte</code>	8 bit unsigned
<code>vchar</code>	8 bit signed
<code>vui16</code>	16 bit unsigned
<code>vsi16</code>	16 bit signed
<code>vui32</code>	32 bit unsigned
<code>vsi32</code>	32 bit signed

Wichtig ist außerdem die Anordnung innerhalb von Strukturen und die Byte-Order, d.h. die Reihenfolge der Bytes innerhalb eines Langworts. Strukturen haben eine rechnerabhängige Packungsdichte, so daß immer die für den Prozessor günstigste Adressierungs- und Zugriffsmethode benutzt werden kann. Die Packungsmethode Ihres Compilers hat daher Folgen auf das Zusammenspiel von LUCA.DLL und Modul sowie auf die Daten, die Sie mit Ihrem Protokoll übertragen möchten. In der LUCA Protocol Factory werden nur Strukturen mit 32 bit benutzt, so daß die Packungsmethode nur bei CPUs mit mehr als 32 bit relevant wird. Achten Sie aber dennoch auf die Packungsmethode innerhalb Ihrer Protokollmodule und Treiber, da ein Übertragen von Strukturen über Netzwerke oder andere Kommunikationswege auch Rechner mit anderer Packungsdichte beinhalten kann. Dort werden dann die `char`, `short`, `int` Werte vielleicht anders interpretiert. Speichern Sie daher alle Strukturen byteweise in Datenpakete, auch wenn es mühsam ist.

1.2.2. Datenblock

Ein Datenblock ist der Baustein, aus dem Messages gebildet werden. Die Struktur `vdblk_t` wird zwar nur intern verwendet, da aber einige Makros und Funktionen mit dieser Struktur arbeiten, soll sie hier dennoch kurz angesprochen werden.



Der Grundgedanke des Datenblockkonzepts ist, daß Module zur Steigerung der Performance so wenig Speicher wie möglich kopieren sollen. Deshalb benutzt nicht jedes Modul einen eigenen Ringpuffer. Andernfalls wäre jedes Modul eine kleine Kopiermaschine. Auch Module, die nur eine einfache Verbindung herstellen (z.B. **ATMODEM**), hätten sonst eine ganze Menge Zeichen zu kopieren und würden somit die Applikation verlangsamen.

Der Inhalt des Datenblocks wird durch den Eintrag `vdb_type` beschrieben, der die verschiedenen LUCA-Messagetypen annehmen kann. Das Macro `VDATA_TYPE` sollte für das Setzen und Abfragen des Datenblocktyps benutzt werden.

1.2.3. Messagetypen

Grundsätzlich werden zwischen Daten- und Kontrollinformationen unterschieden. Messages mit Dateninhalt werden in der API-Schicht durch `Vwrite()` erzeugt und durch `Vread()` in den Anwendungsspeicher kopiert. Außerdem werden die Messages mit unterschiedlicher Priorität behandelt. Datenmessages und das Zeichen für *Ende der Übertragung* (EOF) werden mit geringerer Priorität behandelt als die übrigen Kontrollinformationen.

Die LUCA-Messagetypen im einzelnen:

- **VM_CONNECT** und **VM_CONNECT_CONF** sind für den Verbindungsaufbau zuständig.
- **VM_EOF** und **VM_EOF_CONF** werden beim Verbindungsabbau ausgetauscht.
- **VM_SETPAR**, **VM_SETPAR_CONF** sowie **VM_GETPAR** und **VM_GETPAR_CONF** sind an der Parameterübergabe mit `Vsetpar()` und `Vgetpar()` beteiligt.
- **VM_ERROR** signalisiert Verbindungs- und Protokollfehler, die z.B. durch `VM_SETPAR` entstehen.
- **VM_ABORT** und **VM_DATA_LOST** melden einen Verbindungsabbruch.
- **VM_NONSTDEV** und **VM_PROTOCOLEV** lösen die entsprechenden NONSTD- und PROTOCOL-Ereignisse aus.

1.2.4. Message

1.2.4.1. Messageblock

1.2.4.2. Messageblöcke mit mehreren Teilen

1.2.4.3. Allokieren

1.2.4.4. Daten speichern

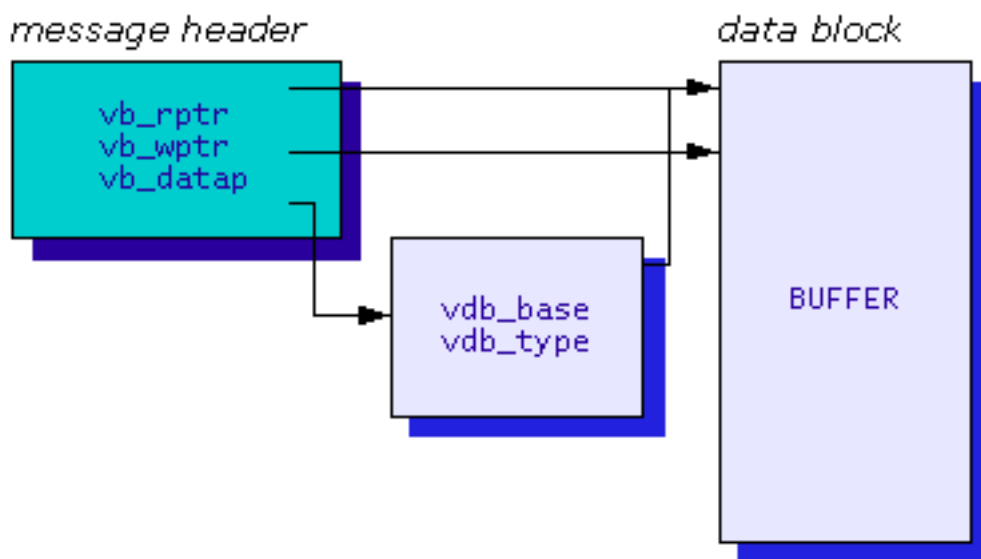
1.2.4.5. Daten auslesen

1.2.4.6. Was passiert, wenn die Anwendung Vread() aufruft?

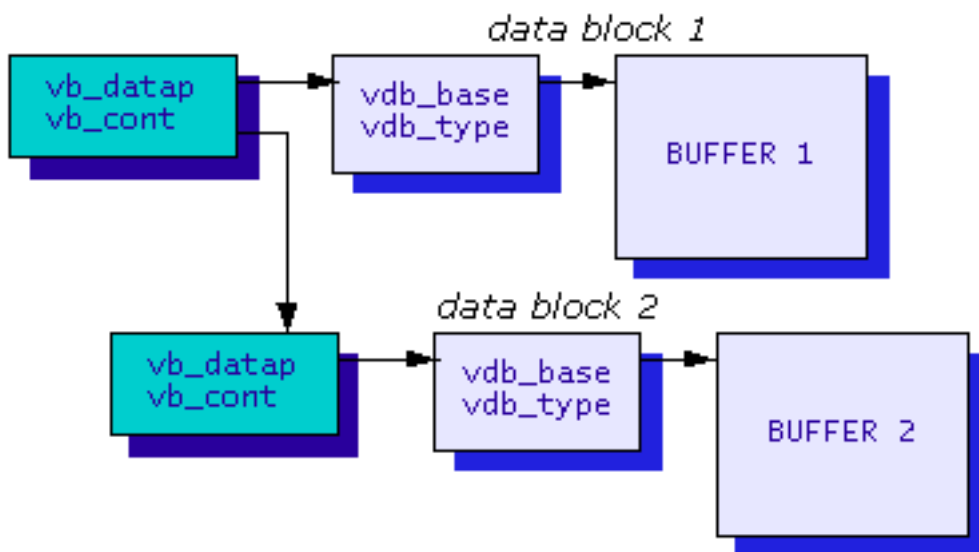
1.2.4.7. Freigeben oder Weitergeben von Messages

1.2.4.1. Messageblock

Ein Messageblock besteht aus einem Datenblock `vdblk_t` und einem Messageheader `vmblk_t`.



Eine komplette **LUCA-Message** besteht aus mindestens einem Messageblock, auf den weitere Messageblöcke folgen können.



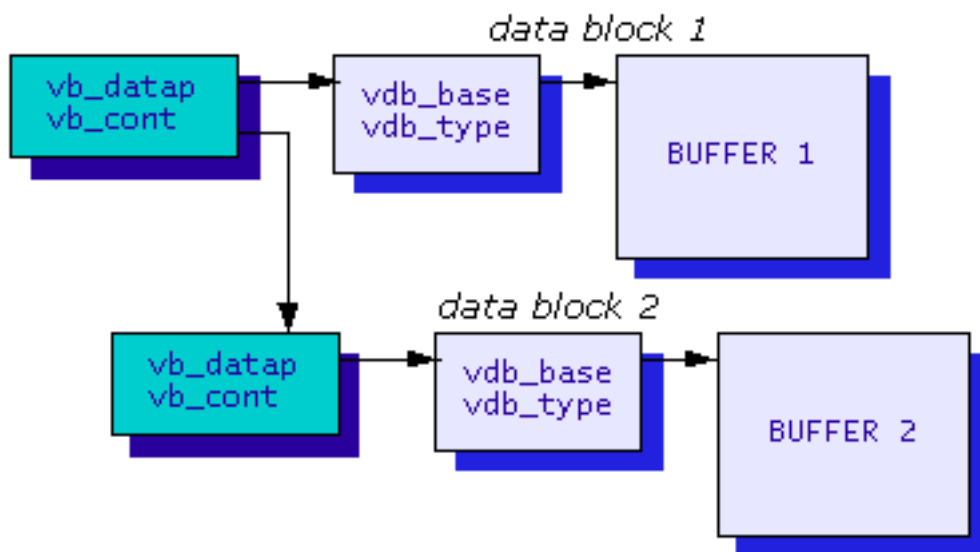
Diese Struktur spielt eine zentrale Rolle in der LUCA Protocol Factory, da sich die gesamte Kommunikation der LUCA-Module untereinander über Messages abspielt. Am anschaulichsten wird diese Tatsache wohl dadurch, daß das Modul **TRACE** fast an jede Stelle im Link Identifier zwischen andere Module gesetzt werden kann, und immer genau die Daten mitprotokolliert werden, die zwischen diesen Modulen ausgetauscht werden. Sie werden später, wenn Sie Ihr erstes Modul testen, mit einem modifizierten **TRACE**-Modul solche Kommunikation noch genauer unter die Lupe nehmen können.

LUCA-Messages enthalten eine (fast) beliebige Anzahl Zeichen oder Bytes. Zu diesen Daten gehören sowohl die eigentlichen Nutzdaten als auch Parameteränderungen und -abfragen durch **Vsetpar** und **Vgetpar** und NONSTD- und PROTOCOL-Events.

Ein Aufruf von **Vsetpar** oder **Vwrite** führt also immer dazu, daß ein Messageblock vom LUCA-Kern angefordert wird und in Richtung Treibermodul durch alle beteiligten Module hindurch geschickt wird. In umgekehrter Richtung werden Daten von einer Schnittstelle, die die Anwendung lesen soll, immer erst vom Treiber in einen Messageblock eingepackt und durch alle beteiligten Module bis zur Anwendung geschickt.

1.2.4.2. Messageblöcke mit mehreren Teilen

Damit nicht zu häufig kopiert werden muß, können Messages aus mehreren Messageblöcken zusammengesetzt sein. Dies ist der Fall bei Protokollen, die paketorientiert arbeiten, z.B. Internet Protokoll oder TCP. In jeder Protokollschicht kommen zu den Rohdaten wieder Kontrolldaten hinzu, die **vor** die Rohdaten gestellt werden. Um eine solche Einkapselung zu ermöglichen, gibt es in LUCA-Messages den Zeiger **vb_cont**, der auf eine Fortführung eines Datenblocks zeigt (s. **vmblok_t**).



Eine komplette LUCA-Message besteht also aus einem ersten Messageblock **vmblok_t** mit einem zugehörigen Datenbereich **vdblok_t**. Es können darauf beliebig viele weitere Messageblöcke inklusiv Datenbereichen folgen.

1.2.4.3. Allokieren

Um eine Message an einen Treiber oder an die Anwendung zu schicken, muß zunächst ein Zeiger auf einen Messageblock vom LUCA-Kern angefordert werden. Die geschieht mit der Funktion **vallocb**.

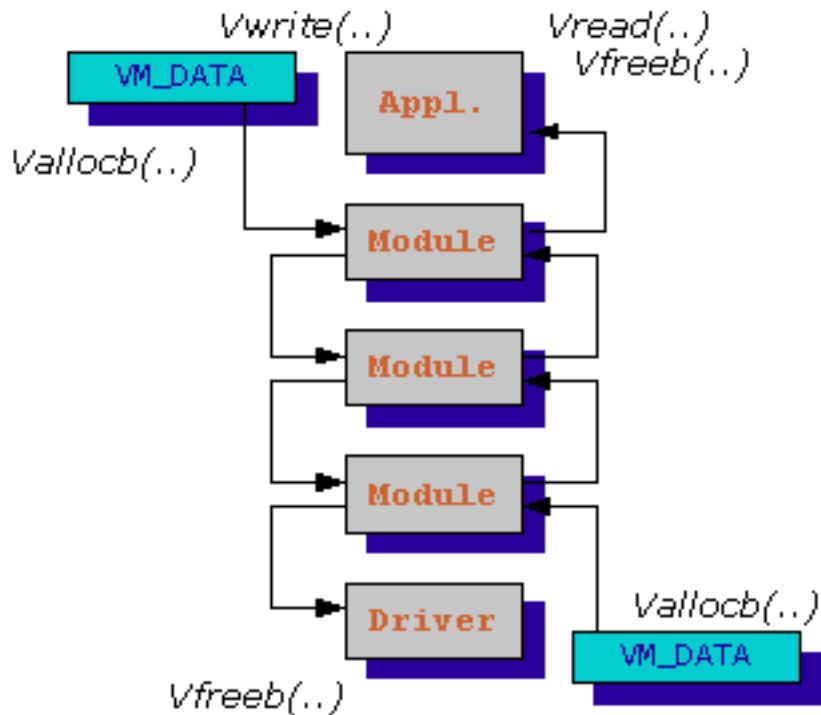
Weil zu jeder Message auch ein Datenblock gehört, wird dieser Datenblock, dessen Größe in Bytes angegeben werden muß, gleich mit angefordert.

```
vmblok_t *mb;
```

```
mb = vallocb(1024,0);
```

Dieses Beispiel liefert einen Zeiger *mb* auf eine Message, die maximal 1024 Bytes Daten enthalten darf.

Es gibt weitere Funktionen in LUCA, die selbst auch Messages anfordern und gleich an ein Modul weiterschicken. Dazu gehören **vputctl** (Senden von Kontrollinformation ohne Daten) und **vfirnonstd** (Auslösen eines NONSTD-Events).



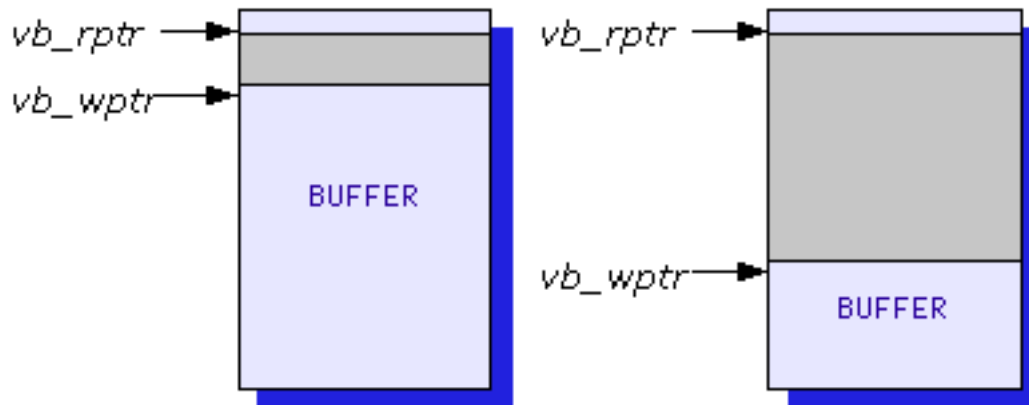
Vom Treiber wird in Leserichtung einmal ein Datenblock vom LUCA-Kern angefordert und von der Anwendung wieder freigegeben, während beim Schreiben ein anderer Datenblock von der Anwendung (in **Vwrite**) angefordert und vom Treiber wieder freigegeben wird. Die Module, die dazwischen in einem Protokollstapel sitzen, müssen nur die Zeiger auf diese Datenblöcke weitergeben.

1.2.4.4. Daten speichern

Zum Ablegen von Daten innerhalb einer Message wird der Zeiger `vb_wptr` verwendet:

```
strcpy(mb->vb_wptr, dialstring);
while (*mb->vb_wptr) mb->vb_wptr++;
*mb->vb_wptr++ = 'A';
*mb->vb_wptr++ = CR;
```

Dieser Zeiger wird mit jedem Zeichen erhöht, so daß die Differenz aus `vb_wptr` und `vb_rptr`, dem Zeiger zum Lesen, genau der Anzahl von Zeichen innerhalb des Datenblocks entspricht.



Der Zeiger `vb_wptr` zeigt daher immer genau auf das nächste beschreibbare Zeichen

Selbstverständlich kann auch der Zeiger erst am Ende eine Operation kurz vor dem Weitergeben an ein anderes Modul um die gesamte Anzahl erhöht werden.

Es darf nie über die Grenze des Datenblocks hinausgeschrieben werden.

1.2.4.5. Daten auslesen

Bekommt ein Modul eine Message, können Daten ausgelesen werden. Das sieht häufig so aus:

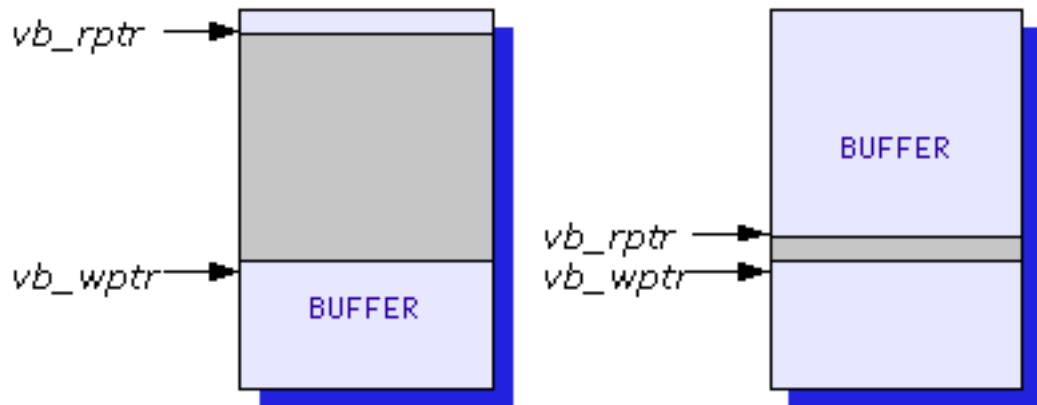
```

vmblok_t *tmp = mb;

while (tmp != NULL) {
    while (tmp->vb_rptr < tmp->vb_wptr) {
        zeichen_verarbeiten(*tmp->vb_rptr++);
    }
    tmp = tmp->vb_cont;
}
vfreemsg(mb);

```

Während der Schreibzeiger immer auf das nächste beschreibbare Zeichen zeigt, zeigt der Lesezeiger `vb_rptr` immer auf das nächste lesbare Zeichen.



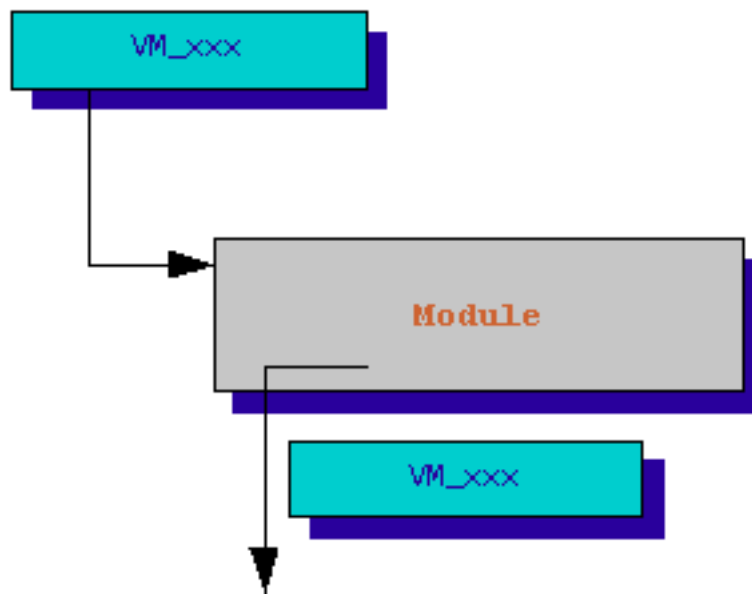
Sind beide Zeiger gleich, sind keine weitere Daten aus der Message auslesbar. Die innere Schleife liest alle Daten innerhalb eines Messageblocks. Die äußere Schleife liest alle Messageblöcke innerhalb der Message (siehe **Datenblöcke mit mehreren Teilen**).

1.2.4.6. Was passiert, wenn die Anwendung `Vread()` aufruft?

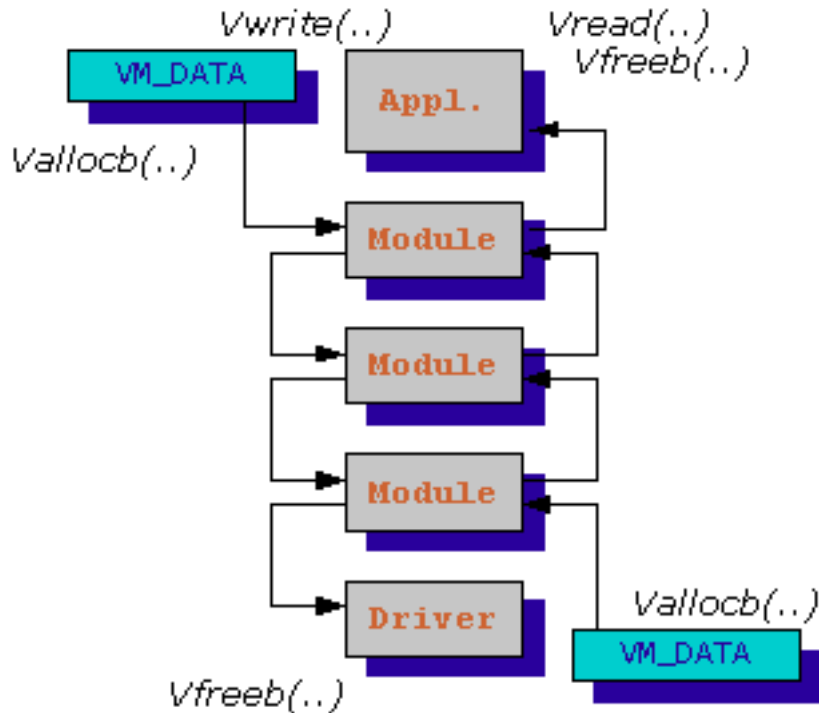
Zu jedem geöffneten Port gehört eine Warteschlange, in der die Messages abgelegt werden, die für die Anwendung bestimmt sind. Beim Aufruf von `Vread()` wird aus dieser Warteschlange der erste Datenblock herausgenommen und in den Anwendungsbereich, oder den Puffer, der bei `Vread()` angegeben wurde, hineinkopiert. Es hängt jetzt vom eingestellten Modus `V_BLOCKING` ab, ob die Zeichen eines kompletten Datenblocks kopiert werden oder nur Teilbereiche. Eine Message entspricht dabei immer einem Telegramm oder einem `Vread`-Aufruf, wenn der Port im Blockmodus (Standardeinstellung) arbeitet.

1.2.4.7. Freigeben oder Weitergeben von Messages

Wenn ein Modul eine Message von einem anderen Modul erhält, kann es entscheiden, ob diese verworfen, zwischengelagert, zurück- oder weitergeben wird.

**Am Ende dürfen jedoch keine Messages übrigbleiben**

Zum Freigeben von Messages dienen die Funktionen **vfreeb** und **vfreemsg** und zum Weitergeben von Messages die Funktion **vputnext**. Zwischenlagern kann man Messages in einer Warteschlange mit **vputq** (s. **Queues**).



1.2.5. Modulinformationen

Bei den folgenden Datentypen handelt es sich um statische Strukturen, die dazu dienen, Informationen über das Modul dem LUCA-Kern bereitzustellen.

Dieser Abschnitt ist nur wichtig, wenn Sie sich näher mit der Technik des Ladens der Module auseinandersetzen möchten. Wahrscheinlich werden Sie diese Tabellen einfach aus einem der Beispiele kopieren und die Felder für Ihre Zwecke anpassen.

Der Einfachheit halber ist es möglich, mehrere Protokollmodule und Treiber in einer DLL bereitzustellen. In der Protocol Factory wird allerdings immer vom Treiber her geladen, also bei `Vopen("MODUL1/MODUL2/TREIBER3")` wird erst nach der DLL "TREIBER3.LUC" gesucht. Die restlichen Module können mit in dieser DLL untergebracht werden. Das Laden von Protokollen, die sowieso nur mit bestimmten Modulen funktionieren, läuft dann schneller ab und man erspart sich, mehrere DLLs (MODUL1.LUC, MODUL2.LUC) auf dem Zielrechner zu installieren.

Exporttabelle

Um ein Modul zu laden, benötigt LUCA nur eine Adresse: die der Exporttabelle. In dieser Tabelle stehen wiederum Adressen von den Modulen, die in der DLL implementiert sind.

Die Exporttabelle hat folgendes Aussehen:

```
vstreamtab_t *implemented_modules[] = {
    &testdrivmodule,
    &testprotomodule,
    0
};
```

Tabelle des Moduls

`vstreamtab_t` ist die einzige Adresse, die ein Modul exportieren muß:

```
vstreamtab_t testprotomodule = {
    &testproto_iread,
    &testproto_iwrite,
```

```
    NULL,  
    NULL  
};
```

Alle anderen Variablen sollten static deklariert sein.

Funktionszeigerinitialisierung

Mit diesen beiden Strukturen werden die Adressen von Modulfunktionen an den LUCA-Kern übergeben:

```
static vqinit_t testproto_iread = {  
    testproto_rput,  
    NULL,  
    testproto_open,  
    testproto_close,  
    &testproto_info, NULL};  
  
static vqinit_t testproto_iwrite = {  
    testproto_wput,  
    NULL,  
    NULL,  
    NULL,  
    &testproto_info, NULL};
```

Die Struktur **vqinit_t** enthält vier Zeiger auf Funktionen in Lese- und Schreibrichtung, die weiter unten unter **module_open**, **module_close**, **module_put** und **module_service** beschrieben werden. Darauf folgt ein Zeiger auf Modulinformationen und ein Zeiger auf Statistikdaten, der z.Zt. nicht verwendet wird.

Modulinformationen

```
static struct vmodule_info testproto_info = {  
    "testproto",  
    0,  
    0,  
    TESTDRIV_HIGH,  
    TESTDRIV_LOW,  
    "1.2",  
    V_MEDIA_PACKAGE_UTIL,  
    V_MEDIA_PACKAGE_UTIL_30,  
    "(C) Module Guru Inc."  
};
```

In dieser Tabelle steht:

- der Name des Moduls
- die minimale und maximale Paketgröße
- die Grenzen für die Flußkontrolle (High- und Low-Watermarks)
- die Versionsnummer
- die Packagezugehörigkeit als Bitmaske
- die Packagezugehörigkeit ab Verion 3.0 als Bitmaske
- ein Text für sonstige Zwecke

Versionsnummern

Die oben erwähnte Versionsnummer (im Beispiel "1.2"), gibt **nicht** die Version des Moduls an, sondern die Version der LUCA.DLL, **ab der das Modul lauffähig ist**.

Packagezugehörigkeit

Es ist nicht möglich, neue Packages in LUCA einzubauen. Hier muß der Eintrag V_MEDIA_PACKAGE_UTIL erfolgen.

Text für sonstige Zwecke

Dieser Text wird vom LUCA-Kern nicht ausgewertet und kann für modulinterne Zwecke, z.B. Copyright oder Versionsnummern, verwendet werden.

1.2.6. Queues

1.2.6.1. Übersicht: Queues

1.2.6.2. Füllstand

1.2.6.3. Daten ablegen

1.2.6.4. Daten auslesen

1.2.6.5. Serviceroutinen

1.2.6.1. Übersicht: Queues

Eine LUCA-Queue (**vqueue_t**) ist der Datentyp, der Messages aufnehmen kann. Daten und Kontrollinformationen können in einer Queue abgelegt und aus ihr wieder nach dem First-In-First-Out-Prinzip (FIFO) ausgelesen werden. Für jeden Port und jedes Modul wird beim Öffnen mit `Vopen()` diese Struktur **zweimal** für Lese- und Schreibrichtung angelegt und beim Schließen wieder freigegeben.

Alle Modulfunktionen erhalten beim Aufruf immer einen Zeiger auf die zugeordnete Queue. Die Funktion `rput(q)` z.B. erhält einen Zeiger auf die Lesequeue.

Für das Arbeiten mit Queues ist nur ein einziger Eintrag der Struktur **vqueue_t** wichtig: `vq_ptr`. Alle anderen Strukturelemente werden über entsprechende Hilfsfunktionen bearbeitet.

Beispiel:

```
testhex_priv *p;

p = (testhex_priv *)malloc(sizeof(testhex_priv));

VWRQ(q)->vq_ptr = (char *)p;
VRDQ(q)->vq_ptr = (char *)p;
```

In diesem Ausschnitt wird den Lese- und Schreibqueues des Beispielmoduls ein Speicherbereich für eigene Zwecke zugeordnet. In einer der anderen Modulfunktionen kann dann über

```
testhex_priv *p = (testhex_priv *)q->vq_ptr;
```

auf diesem Bereich zugegriffen werden. Dieses Verfahren entspricht einer Vererbung in der objektorientierten Programmierung.

1.2.6.2. Füllstand

Der Füllstand einer Queue wird intern durch Flags markiert, so daß beim Auslesen aus einer Queue und Hineinschreiben in eine Queue diese Flags abgeglichen werden können.

Wenn Sie in eine Queue hineinschreiben, sollte immer zuerst mit **vcanput** überprüft werden, ob dies möglich ist. Es gibt aber Fälle, in denen eine Queue überhaupt keine Füllstandskontrolle besitzt, nämlich immer dann, wenn es sich um Module handelt, die keine Service-Funktion besitzen. Dieser Typ von Modulen kann Daten nur umwandeln, ein- oder auspacken. Solche Module haben keine Möglichkeit, in die Flußkontrolle einzugreifen.

Als Beispiel dient hier unser Testtreiber, der Daten an die Anwendung schickt, solange die Anwendung lesen kann:

```
while (vcanput(q->vq_next) && !p->delay) {
    sendbuf(q);
}
```

Die Routine `sendbuf(q)` füllt eine Message mit Daten und schickt diese an das nächste Modul in Richtung Anwendung (im Protokollstack nach oben).

1.2.6.3. Daten ablegen

Mit **vputq** wird eine Message in der zum Modul gehörenden Queue gespeichert. Der LUCA-Kern sorgt in diesem Fall dafür, daß ein Aufruf von **vcanput** den Füllstand der Queue nach dem Speichern von Daten zurückliefert. Im Prinzip können allerdings beliebig viele Aufrufe **vputq** Messages in eine Queue speichern, was dazu führen kann, daß irgendwann ein Speichermangel auftritt. Das Modul muß also in diesem Fall selbst dafür sorgen, daß die Queue nicht überläuft. Die einfachste Anwendung von **vputq** ist als Funktion für **module_put** zu dienen. Dieses Verfahren wird im Kapitel **Protokolle mit Flußkontrolle** beschrieben. In diesem Fall sorgt ein anderes Modul in der entsprechenden Serviceroutine für die Flußkontrolle.

Daten und Kontrollinformationen können mit der Funktion **vputnext** an das nächste Modul weitergegeben werden. Als Beispiel soll hier wieder unser Testtreiber dienen:

```
while (vcanput(q->vq_next) && !p->delay) {
    mb = vallocb(1024,0);
    fill_buffer(mb);
    vputnext(q,mb);
}
```

Hier werden solange Messages vom LUCA-Kern angefordert, mit Daten gefüllt und an das nächste Modul mit **vputnext** weitergereicht, wie das nächste Modul noch Platz für Daten in seiner Queue hat.

Man könnte sich an dieser Stelle statt dem Aufruf von **vputnext** auch einen Aufruf von **vputq(q->vq_next,mb)** vorstellen, der die Message in der Queue des nächsten Moduls ablegt. Dieser Vorgang bleibt aber dem nächsten Modul vorbehalten, das eventuell überhaupt keine Flußkontrolle benötigt und die Daten einfach nur weiterreicht (z.B. das **ATMODEM** Modul).

Bitte achten Sie darauf, daß nie Daten in Messages bearbeitet werden, die schon mit **vputq** oder **vputnext** abgelegt oder weitergereicht wurden. Die Message *gehört* einem Modul nicht mehr, sobald eine dieser Funktionen benutzt wurde.

1.2.6.4. Daten auslesen

Um Messages aus einer Queue zurückzubekommen, wird die erste Message nach dem FIFO-Prinzip mit **vgetq** aus der Queue ausgelesen. Dazu muß allerdings erwähnt werden, daß das FIFO-Prinzip in dieser Form nicht ganz die richtige Bezeichnung ist. Jedem Messagetyp ist eine Priorität zugeordnet, die es ermöglicht die Messages beim Zugriff auf die Queue in einer bestimmten Reihenfolge abzuarbeiten. Als Protocol Factory-Anwender müssen Sie sich darum nicht kümmern, da diese Prioritäten Einfluß auf die zeitliche Abfolge der LUCA-Grundfunktionen haben. Beispielsweise ist ein Aufruf von **Vsetpar**, der **VM_SETPAR** Messages erzeugt, trotz aktuell anstehender Lesedaten möglich. Auf der anderen Seite erzeugt ein Aufruf von **Vclose** zwar eine **VM_EOF**-Message, diese wird aber erst **nach** dem letzten Datenpaket von den Modulen ausgewertet.

1.2.6.5. Serviceroutinen

Die Serviceroutinen der Module werden vom LUCA-Kern aufgerufen, solange Daten oder Kontrollinformationen in Form von LUCA-Messages von einem Modul zum nächsten transportiert werden sollen. Kein Modul darf beliebig lange Rechenzeit in Anspruch nehmen, daher werden vom Scheduler des LUCA-Kerns nacheinander die Serviceroutinen der einzelnen Module abgearbeitet.

Solche Serviceroutinen kann man sich am besten als Pumpe vorstellen. Die Pumpe kann eingeschaltet werden, wenn Wasser benötigt wird. Solange Wasser vorhanden ist **und** das Gefäß nicht voll ist soll gepumpt werden. Sobald wieder Wasser benötigt wird **oder** wieder Wasser vorhanden ist, wird erneut gepumpt.

Im Sourcecode sieht das so aus:

```
static int testprot_rsvr(vqueue_t *q)
{
    vmbk_t *mb;

    while (vcanput(q->vq_next)) {
        if ((mb = vgetq(q)) != NULL)
            vputnext(q,mb);
        else
            break;
    }
    return 0;
}
```


1.3. Ein einfaches Beispielmodul

- 1.3.1. Taten mit Daten
 - 1.3.2. Die Exporttabelle
 - 1.3.3. Die open-Funktion
 - 1.3.4. Die close-Funktion
 - 1.3.5. Die wput-Funktion
 - 1.3.6. Die rput-Funktion
 - 1.3.7. Umwandeln der Daten
 - 1.3.8. Einfache Parameter
 - 1.3.9. Kompilieren und Linken des Moduls
 - 1.3.10. Test- und Tracemöglichkeiten
-

1.3.1. Taten mit Daten

Dieses Kapitel beschreibt, wie man ein einfaches Modul erstellt, daß nur Daten verändert. Zu solcher Art von Modulen gehören z.B. Meßgerätesteuerungen, Autologin, HDLC-Framing usw., und es wäre eigentlich übertrieben, diese Module Protokollmodule zu nennen.

Unser Beispielmodul soll Lesedaten, die von der Treiberseite her kommen, als Hexadezimalausgabe an die Applikation weiterleiten. Um die Parameterübergabe darzustellen, definieren wir einen Parameter LEN, mit dem die Zeilenlänge als Anzahl der ausgegebenen Bytes pro Zeile angegeben werden kann. Dieses Modul kann man später zum Testen verwenden, falls gerade keine andere Tracemöglichkeit wie Debugger o.ä. vorhanden ist.

Das gesamte Projekt besteht aus drei Sourcedateien:

```
lucapf.c
testhex.c
lucapf.h
```

Die Funktionalität des Moduls ist größtenteils in testhex.c implementiert.

1.3.2. Die Exporttabelle

Die Datei lucapf.c wird benutzt, um die Funktion *ModuleInit()* an den LUCA-Kern zu exportieren, damit LUCA die Zeiger auf die Exporttabelle finden kann. Andersherum übergibt der Kern an die Modul-DLL einen Zeiger auf wichtige Kernfunktionen.

```
#include <lucapf.h>
extern struct _vstreamtab testhexmodule;
static struct _vstreamtab *implemented_modules[] = {
    &testhexmodule,
    NULL
};
```

Für die bessere Übersicht wird hier nur unser Beispielmodul **testhex** eingebunden.

Weiterhin benötigen wir den Zeiger auf die Kernfunktionen:

```
_vmdkfuncs *vmdk = 0;
```

Die Funktion *ModuleInit()* tauscht nun diese beiden Zeiger aus:

```
VEXPORT char * VDLLCTYPE
ModuleInit(_vmdkfuncs *vt, struct _vstreamtab ***mtable)
```

```
{
    vmdk = vt;
    *mtable = &implemented_modules[0];
    return 0;
}
```

Mehr sollte in `lucapf.c` nicht passieren.

In der jeweiligen Datei für ein Modul stehen die Modulinformationstabellen, wobei vorher für die Deklaration notwendige Dateien wie `lucapf.h` und `luca.h` eingebunden werden müssen:

```
#include <lucapf.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include <luca/luca.h>

static int testhex_open(vqueue_t *q, int mode);
static int testhex_close(vqueue_t *q);
static int testhex_wput(vqueue_t *q, vmbblk_t *mb);
static int testhex_rput(vqueue_t *q, vmbblk_t *mb);

static struct _vmodule_info testhex_info = {
    "testhex",
    0,
    0,
    0,
    0,
    "1.1",
    V_MEDIA_PACKAGE_UTIL,
    V_MEDIA_PACKAGE_UTIL_30,
    "(C) Langner GmbH"
};
```

Das Modul heißt **testhex**, kann mit allen LUCA-Packages benutzt werden und das Modul läuft ab LUCA-Version 1.1.

Bei Modulen dieses Typs sind nur einfache put-Funktionen notwendig, daher werden die anderen Funktionszeiger mit NULL initialisiert:

```
static struct _vqinit testhex_iread = {
    testhex_rput,
    NULL,
    testhex_open,
    testhex_close,
    &testhex_info, NULL};

static struct _vqinit testhex_iwrite = {
    testhex_wput,
    NULL,
    NULL,
    NULL,
    &testhex_info, NULL};
```

Zum Schluß kommt noch die Funktionstabelle des Moduls, die an den LUCA-Kern exportiert wird:

```
struct _vstreamtab testhexmodule = {
    &testhex_iread,
    &testhex_iwrite,
    NULL,
    NULL
};
```

Im nächsten Abschnitt werden die einzelnen Funktionen `open`, `close` `wput` und `rput` besprochen.

1.3.3. Die `open`-Funktion

Diese Funktion wird vom LUCA-Kern aufgerufen, sobald in einem `Vopen`-Aufruf der Modulname auftaucht. Wichtig dabei ist, daß LUCA die Module von der Treiberseite her initialisiert, also im Link Identifier von rechts gelesen.

```
static int
testhex_open(vqueue_t *q, int mode)
{
    testhex_priv *p;
```

Jedes Modul kann prinzipiell als Treiber oder als Protokoll geladen werden, was durch den Parameter *mode* signalisiert wird. Unser `testhex`-Modul macht jedoch als Treiber keinen Sinn, weshalb wir sicherstellen, daß es nur als Protokollmodul geladen werden kann. Die folgende Zeile erzeugt einen Fehlerzustand, wenn das `testhex`-Modul fälschlich als Treiber geladen wird:

```
    if (mode != MODULE_OPEN) return VE_MODULE_FAILED;
```

Wenn ein Modul nicht korrekt geöffnet wird, kann es dies durch einen der `VE_`-Fehlercodes an die Applikation zurückmelden. Dieser Fehlercode wird direkt der Rückgabewert von `Vopen()`.

Nun benötigt das Modul einige Bytes an Speicher, die zuvor als `testhex_priv`-Struktur deklariert wurden:

```
    p = (testhex_priv *)malloc(sizeof(testhex_priv));
    if (!p) return VE_NO_MEMORY;

    p->len      = 16;
    p->lineno   = 0;
    p->charcount = 0;
```

Nach dem Initialisieren der Struktur wird der Zeiger `p` auf die modulinternen Daten in der Queue-Struktur hinterlegt, da alle Modulfunktionen immer mit dem entsprechenden Zeiger auf die Queue aufgerufen werden und nicht auf die modulinternen Daten.

```
    VWRQ(q)->vq_ptr = (char *)p;
    VRDQ(q)->vq_ptr = (char *)p;
    return 0;
}
```

Bitte beachten Sie, daß die `open`-Funktion (trotz ihres Namens) keine Verbindung öffnet. Außerdem wird diese Funktion genau einmal beim Öffnen und die `close`-Funktion genau einmal beim Schließen *einer Instanz des Moduls* aufgerufen, wenn es beim Öffnen nicht zu einem Fehler kam. Das Freigeben von Speicher wird dadurch vereinfacht.

1.3.4. Die `close`-Funktion

Die `close`-Funktion wird beim Freigeben des LUCA-Ports aufgerufen, z.B. nach einem `Vclose()`. Diese Funktion hat ebensowenig wie die `open`-Funktion mit Verbindungsauf- oder abbau zu tun.

Es kann nach einem erfolgreichen Laden mehrerer Module trotzdem nicht zum Verbindungsaufbau kommen, weil die Parameterübergabe der Module erst nach den `open`-Aufrufen erfolgt. Stellen Sie sich z.B. einen Link Identifier vor, der `ATMODEM,LINE=PZX/ASYNC/COM:3` lautet. LUCA öffnet zuerst das Modul `COM`, dann `ASYNC`, dann `ATMODEM`. Daraufhin bekommt das Modul `COM` den Parameter `Adresse=3` und das Modul `ATMODEM` den Parameter `LINE=PZX`. Weil der Wert `PZX` ungültig ist, führt die Parameterverhandlung zu einem Fehler und die `close`-Funktionen der einzelnen Module werden in umgekehrter Reihengfolge aufgerufen erst `ATMODEM`, dann `ASYNC`, zuletzt `COM`.

Für die Implementierung unseres `testhex`-Moduls ist es daher wichtig, daß in der `close`-Funktion nur der Speicher freigegeben wird, den wir beim Öffnen des Moduls angefordert haben:

```
static int
```

```
testhex_close(vqueue_t *q)
{
    testhex_priv *p = (testhex_priv *)q->vq_ptr;

    free(p);
    return 0;
}
```

1.3.5. Die wput-Funktion

Unser Beispielmódul soll die Daten in Schreibrichtung nicht verändern und erhält deshalb eine sehr einfache Funktion zum Verarbeiten der Messages in Schreibrichtung. Hier benötigen wir nicht einmal einen Zeiger auf die privaten Daten des Moduls:

```
static int
testhex_wput(vqueue_t *q, vmblock_t *mb)
{
    switch (VDATA_TYPE(mb)) {
    case VM_SETPAR:
    case VM_GETPAR:
        vdispatchparam(q, mb, testhex_params);
        return 0;

```

Der einzige Messagety, den wir hier verarbeiten wollen, ist derjenige für Parameterabfragen oder -änderungen. Dafür stellt LUCA die gesonderte Funktion **vdispatchparam** zur Verfügung, die die Tabelle `testhex_params` abarbeitet. Alle anderen Messages werden weitergereicht, da das Modul `testhex` als Protokollmodul arbeitet und im Link Identifier nie ganz rechts stehen kann.

```
        default:
            break;
    }
    vputnext(q, mb);
    return 0;
}
```

Grundsätzlich empfiehlt es sich, am Ende einer put-Funktion immer den Standardfall (`vputnext`) zu benutzen und bei *Verbrauch* der Message, z.B. bei Freigabe des Speichers durch **vfreemsg**, die put-Funktion zu verlassen. Im Beispiel wird die Message von **vdispatchparam** freigegeben.

1.3.6. Die rput-Funktion

In dieser Funktion sollen ankommende Datenpakete in einen Hexdump umgewandelt werden. Falls wir die Funktionsweise des Moduls später noch ändern sollten, ist es besser, die eigentliche Ausgabe der Daten und die Freigabe des Messageblocks durch **vfreemsg** in einer anderen Funktion zu erledigen.

Daher ist die `rput`-Funktion genauso kurz wie die `wput`-Funktion:

```
static int
testhex_rput(vqueue_t *q, vmblock_t *mb)
{
    switch (VDATA_TYPE(mb)) {
    case VM_DATA:
        return hex_chg(q, mb);
    default:
        break;
    }
    vputnext(q, mb);
    return 0;
}
```

An dieser Stelle sei nochmals darauf hingewiesen, daß LUCA-Module nicht in Endlosschleifen auf Daten warten dürfen. Module bekommen Daten über eine put-Funktion und müssen diese Funktion schnellstmöglich verlassen, um anderen Modulen Rechenzeit zukommen zu lassen.

1.3.7. Umwandeln der Daten

Mit der folgenden Funktion wird eine komplette Message in einen Hexdump umgewandelt.

Ein häufig auftretendes Problem besteht darin, die Größe des neu entstehenden Datenblocks auszurechnen. In unserem Beispiel werden immer Datenblöcke mit der dreifachen Byteanzahl pro Zeile (plus einige Bytes für Adresse und Zeilenende) generiert. Ist eine Zeile voll, wird die Zeile an die Applikation geschickt.

Zuerst werden einige Variablen und der Zeiger auf die modulinternen Daten initialisiert:

```
static int
hex_chg(vqueue_t *q, vmblok_t *mb)
{
    testhex_priv *p = (testhex_priv *)q->vq_ptr;
    vmblok_t *blk = NULL;
    unsigned char *w;
    int done = 0;
```

Die nächsten zwei Schleifen durchlaufen alle Messageblöcke einer Message und alle Daten eines Messageblocks. Die Daten werden dann in einen Hexdump umgewandelt:

```
while (mb != NULL) {
    w = mb->vb_rptr;
    while (w < mb->vb_wptr) {
```

Jetzt benötigen wir einen Puffer, in den eine Zeile der Länge *len* hineinpaßt. Vor der Zeile benötigen wir zusätzlich 8 Zeichen für die Adresse und am Ende der Zeile je ein Zeichen für CR, LF und eine Null als Sicherheit (wegen `sprintf`, Anm. d. Redaktion):

```
if (blk == NULL) {
    blk = vallocb(8 + p->len + 2 + 1, 0);
    if (blk == NULL) {
        vfireerror(VWRQ(q), VE_NO_MEMORY, "", mb);
        return -1;
    }
}
```

Sollte ein Speicherproblem entstehen, sorgt `vfireerror` für eine entsprechende Meldung.

Am Anfang einer Zeile soll eine Zeilennummer eingefügt werden:

```
sprintf(blk->vb_wptr, "%08X", p->lineno);
while (*blk->vb_wptr) blk->vb_wptr++;
}
```

Der Schreibzeiger `vb_wptr` steht nach der `while`-Schleife auf dem nächsten beschreibaren Zeichen.

Nach dem Anfang der Zeile kommen nun die einzelnen Bytes:

```
sprintf(blk->vb_wptr, " %02X", ((unsigned int)(*w++)) & 0xff);
while (*blk->vb_wptr) blk->vb_wptr++;
```

Sollte die Zeile länger als `p->len` werden, wird die komplette Zeile an das nächste Modul weitergegeben. `blk` wird auf `NULL` gesetzt, damit beim nächsten Byte wieder Speicher für eine neue Zeile angefordert wird. Die Message, auf die der Zeiger `blk` zeigt, wird vom nächsten Modul mit `vputnext` verarbeitet und dort freigegeben.

```
if (++done == p->len) {
    sprintf(blk->vb_wptr, "\r\n");
    while (*blk->vb_wptr) blk->vb_wptr++;
    p->lineno+=done;
    done = 0;
    vputnext(VRDQ(q), blk);
    blk = NULL;
}
}
```

```
mb = mb->vb_cont;
}
```

Zum Schluß müssen wir darauf achten, die restliche Zeile an die Applikation zu schicken, wenn noch Daten ausstehen. Außerdem wird die komplette, umgewandelte Message freigegeben.

```
if (blk) {
    sprintf(blk->vb_wptr, "\r\n");
    while (*blk->vb_wptr) blk->vb_wptr++;
    vputnext(VRDQ(q), blk);
    p->lineno += done;
}
vfreemsg(mb);
return 0;
}
```

1.3.8. Einfache Parameter

Das Verfahren der Parameterübergabe soll hier an einem kleinen Beispiel verdeutlicht werden. Die Anwendung öffnet einen Port mit dem Link Identifier

```
Vopen("ASYNC/COM:2,SPEED=9600");
```

Daraufhin findet zwischen dem Modul COM und dem LUCA-Kern folgende Kommunikation statt:

Kern	COM
VM_SETPAR SPEED 9600	---->
<---	VM_SETPAR_CONF
VM_SETPAR SUBADDR 2	---->
<---	VM_SETPAR_CONF
VM_CONNECT	---->
<---	VM_CONNECT_CONF

Die Parameterübergabe der Parameter des Link Identifiers erfolgt also immer vor der VM_CONNECT Message, während ein Aufruf von Vsetpar(port,"SPEED=9600") nach dem VM_CONNECT Signal ausgeführt wird. Das Senden von VM_CONNECT zeigt immer den Abschluß von **Vopen** an, erst danach hat die Anwendung überhaupt erst die Möglichkeit, ein Vsetpar durchzuführen.

Die Messages VM_GETPAR und VM_GETPAR_CONF funktionieren ähnlich, jedoch brauchen Sie sich um Details nicht zu kümmern. Wie Sie oben bereits gesehen haben, gibt es eine Kernfunktion, die die Auswertung der VM_SETPAR- und VM_GETPAR-Messages übernimmt:

```
vdispatchparam(q, mb, testhex_params);
```

vdispatchparam liest die Tabelle testhex_params solange, bis ein passender Parameter gefunden wurde. Daraufhin wird die dazugehörige Funktion in der Tabelle aufgerufen:

```
static vparams_t TESTHEX_PARAMS[] = {
    "LEN", testhex_len,
    NULL, NULL
};
```

Im Fall von Vsetpar(port,"LEN=23") oder Vopen("TESTHEX,LEN=23/ASYNC/COM:3") wird die Funktion testhex_len unabhängig davon aufgerufen, ob bereits eine Verbindung besteht oder nicht.

Es gibt insgesamt vier Fälle, die auftreten können:

- Ändern eines Parameters durch Vsetpar oder Vopen
- Abfragen eines Parameters mit Vgetpar("LEN")
- Abfragen möglicher Werte mit Vgetpar("LEN?")
- Abfragen aller möglichen Parameter mit Vgetpar("") oder Vgetpar("+").

Die Parameterfunktion im testhex-Modul sieht folgendermaßen aus:

```
static char *
```

```
testhex_len(vqueue_t *q, char *val, int function)
{
    testhex_priv *p = (testhex_priv *)q->vq_ptr;
    static char buf[40];
    switch (function) {
```

Dieser Funktionskopf kommt Ihnen sicherlich schon bekannt vor, da hier zunächst nur ein Zeiger auf die modulinternen Daten angelegt wird.

Jetzt kommen drei der vier möglichen oben genannten Fälle:

```
    case V_GETPAR:
        sprintf(buf, "%d", p->len);
        return buf;
    case V_SETPAR:
        if (strlen(val) > 20) return NULL;
        if (!isdigit(*val)) return NULL;
        p->len = atoi(val);
        return val;
    case V_GET_POSSIBLE_PAR:
        return V_POS_INTEGER;
    default:
        return NULL;
}
}
```

Dabei gibt es mehrere verschiedene Rückgabewerte:

- NULL für Fehlfunktion
- den Zeiger *val* für korrekte Zuweisung bei V_SETPAR und andere Textstrings, z.B. bei V_GETPAR den aktuellen Wert
- bei V_GET_POSSIBLE_PAR die möglichen Werte

Um den vierten Fall, das Auflisten, brauchen Sie sich nicht zu kümmern, weil dies bereits durch **vdispatchparam** erledigt wird.

1.3.9. Kompilieren und Linken des Moduls

Um eine DLL *testhex.luc* zu erzeugen, müssen sie die Kompileroption für Multithreaded-Umgebung benutzen, da LUCA mehrere Threads benutzt.

Als Ausgabedatei müssen Sie **testhex.luc** einstellen, weil der LUCA-Kern beim Laden von externen Modulen diese Endung benutzt.

Kopieren Sie anschließend die entstandene DLL in das Verzeichnis, in dem sich auch die Datei LUCA.DLL befindet, oder in ein anderes Verzeichnis, das für DLLs durchsucht wird. Wenn Sie Microsoft Developer Studio verwenden, können Sie entweder die Datei kopieren (nicht empfohlen), oder Sie benutzen das Debug-Verzeichnis, in dem die DLL beim Linken erzeugt wird, als das Arbeitsverzeichnis. Häufig gibt es Probleme dadurch, daß normalerweise *per Hand* kopiert wird, dann aber irgendwann das Kopieren vergessen wird.

1.3.10. Test- und Tracemöglichkeiten

Ein großes Problem beim Entwickeln von Protokollmodulen ist die Zeitabhängigkeit. Breakpoints und Single-Stepping verändern das Verhalten eines Moduls dermaßen, daß alles anders läuft, als erwartet wird. Manchmal treten Probleme sogar nur so selten auf, daß man tagelang warten muß, bis ein Fehler nochmals auftritt. Deshalb liefern wir mit der Protocol Factory einige Module mit, die das Testen erleichtern.

Der einfachste Vertreter ist das **ECHO** Treibermodul, mit dem Sie nun Ihr *testhex*-Modul ohne Internet-, Asynchron- oder ISDN-Verbindung ausprobieren können. Wenn das Modul mit ECHO zusammen funktioniert, wird es ohne Anpassungen mit allen anderen Modulen zusammenspielen. ECHO bietet zusätzlich den Vorteil, die Daten ohne Datenverlust oder -veränderung übertragen zu können. Selbst ein Nullmodem an einem COM-Port im PC neigt dazu, manchmal hartnäckig Zeichen zu verlieren (z.B. weil der Stecker abgerutscht ist oder kein 16550 UART benutzt wird).

Eine weitere Testhilfe bietet das MTRACE-Modul, das mit der Protocol Factory im Sourcecode geliefert wird. Es ermöglicht eine Ausgabe aller Messages, die zwischen zwei Modulen ausgetauscht werden. Im

Unterschied zum normalen **TRACE** Modul werden die Messages im Debug-Fenster des Developer Studios ausgegeben anstatt in einer Datei. Damit wird Ihnen die Möglichkeit gegeben, während eines Programmlaufs den Protokolltrace *online* mitzulesen.

Geben Sie als Link Identifier z.B. **MTRACE/testhex/echo** oder **testhex/MTRACE/echo** ein, um zu sehen, welche Messages zwischen den Modulen ausgetauscht werden.

1.4. XONXOFF: Ein Beispielprotokollmodul mit Flußkontrolle

1.4.1. Die Serviceroutinen

1.4.2. Eine einfache Serviceroutine

1.4.3. Ein- und Ausschalten der Queue

1.4.4. Senden der Steuerinformation

1.4.5. Füllstände und -grenzen

1.4.1. Die Serviceroutinen

Serviceroutinen sind im Prinzip nichts anderes als Interruptroutinen, die Sie eventuell von früheren Projekten her kennen, wenn Sie Treiber programmiert haben. Eine Serviceroutine wird vom Scheduler des LUCA-Kerns immer dann aufgerufen, wenn sich durch Auslesen oder Hineinschreiben in eine Queue die Datenmenge so verändert, daß erneutes Auslesen oder Hineinschreiben notwendig wird.

Dazu ein Beispiel: Ein UART, der für die serielle Datenübertragung verantwortlich ist, löst eine Unterbrechungsanforderung aus, sobald das Senderegister leer ist. In der entsprechenden Routine wird nun der Sendepuffer abgefragt. Sollten noch Zeichen zu senden sein, wird das nächste Zeichen an den UART übergeben. Nach einiger Zeit sind keine Daten zu verschicken und der Sendepuffer wird leer. Wird jetzt wieder in den Sendepuffer geschrieben, muß genau die gleiche Routine wieder ein Datum aus dem Sendepuffer herausnehmen und an den UART übergeben, wie zuvor während der Unterbrechungsanforderung.

Es gibt also immer zwei Fälle, die dazu führen, daß die Serviceroutine aufgerufen wird:

- Das nächste Modul war vorher blockiert und kann wieder Daten empfangen. Dieser Fall entspricht dem UART, der durch die Unterbrechungsanforderung signalisiert, daß er wieder bereit ist, Daten entgegenzunehmen.
- Die Queue war vorher leer und wird gefüllt. Die Anwendung oder ein übergeordnetes Modul möchte wieder Daten schreiben.

Im Beispielprogramm `xonoff.c` wird dieses Verhalten mit einem XON/XOFF Handshake simuliert. Sobald auf der Empfängerseite die Queue zu voll wird, schickt das Modul ein XOFF-Zeichen (Ctrl-S) an die Gegenseite. Können wieder Daten gesendet werden, wird ein XON-Zeichen (Ctrl-Q) zur Gegenseite geschickt. Der Scheduler im LUCA-Kern erleichtert dabei die Realisierung enorm, da die Serviceroutinen nur in den entsprechenden Momenten aufgerufen werden.

Im nächsten Abschnitt werden die wichtigsten Quellcodeteile dieses Moduls besprochen.

1.4.2. Eine einfache Serviceroutine

In Senderichtung fällt die Serviceroutine besonders einfach aus, da hier keine Daten analysiert werden müssen. Das einzige, was diese Routine tun muß, ist, Daten zu kopieren. Daher wird diese Art Schleife auch Datenpumpe genannt:

```
static int
xonxoff_wsvr(vqueue_t *q)
{
    vmbk_t *mb;

    while (vcanput(q->vq_next)) {
        if ((mb = vgetq(q)) != NULL) {
            vputnext(q, mb);
        }
        else break;
    }
    return 0;
}
```

Solange Daten zur Anwendung geschickt werden können (**vcanput**), wird versucht, aus der Sendequete mit **vgetq** eine Message auszulesen. Diese Message wird sogleich per **vputnext** an das nächste Modul weitergereicht.

Diese Schleife darf auf keinen Fall umgekehrt benutzt werden:

```
while ((mb = vgetq(q)) != NULL) {...
```

Dadurch würde jedesmal eine Message aus der Queue entfernt werden, obwohl u.U. überhaupt nichts gesendet werden darf.

Wichtig ist hierbei, daß der LUCA-Kern beim Weiterreichen von Messages keine Kontrolle über Art oder Größe der Messages hat. Eine Queue könnte also z.B. Flowcontrolgrenzen von 1K und 2K einstellen, wobei dennoch mit **vputq** Datenblöcke mit 8K in die Queue geschrieben werden können. Es handelt sich bei diesen Grenzangaben nicht um absolute Maximalangaben, sondern nur um Richtwerte.

1.4.3. Ein- und Ausschalten der Queue

Wie kann nun die eben dargestellte Datenpumpe durch XON- oder XOFF-Zeichen gesteuert werden? Die Antwort liegt in den Makros **venableok**, **vnoenable** und der Funktion **vqenable**. Eingeschaltet (enabled) ist eine Queue immer dann, wenn die Pumpe Daten verarbeiten soll. Wie wir am Anfang dieses Abschnitts gesehen haben, kann das Einschalten durch mehrere Ereignisse hervorgerufen werden. Ein neues Ereignis wird nun der Empfang des Zeichens XON. Die Datenpumpe **muß** in diesem Fall weiterlaufen (**venableok**, **vqenable**). Der andere Fall, also der Empfang eines XOFF-Zeichens, soll die Datenpumpe abschalten (**vnoenable**).

```
static int
xonxoff_rput(vqueue_t *q, vmblok_t *mb)
{
    vmblok_t *tmp;
    unsigned char *w,*r;

    if (VDATA_TYPE(mb) == VM_DATA) {
        tmp = mb;
        while (tmp != NULL) {
            r = w = tmp->vb_rptr;
            while (r < tmp->vb_wptr) {
                if (*r == XON) {
                    venableok(VWRQ(q));
                    vqenable(VWRQ(q));
                } else if (*r == XOFF) {
                    vnoenable(VWRQ(q));
                } else *w++ = *r;
                r++;
            }
            tmp->vb_wptr = w;
            tmp = tmp->vb_cont;
        }
        if (mb->vb_wptr == mb->vb_rptr) {
            vfreemsg(mb);
            return 0;
        }
    }
    vputq(q,mb);
    return 0;
}
```

In dieser Routine wird eine Datenmessage komplett nach XON/XOFF-Zeichen durchsucht und die Steuerzeichen werden aus dem Datenpuffer herausgenommen. Leere Messages werden verworfen (**vfreemsg**), während alle anderen in die Queue gepackt werden (**vputq**).

1.4.4. Senden der Steuerinformation

Abschließend werfen wir noch ein Blick auf die Empfangsserviceroutine. Hier müssen immer dann XOFF-Zeichen geschickt werden, wenn die Anwendung keine Daten abholt (**vcanput**). Sobald die eigene

Warteschlange leergelaufen ist (**vgetq** liefert NULL), muß der Gegenseite dies durch ein XON-Zeichen mitgeteilt werden.

```
static int
xonxoff_rsvr(vqueue_t *q)
{
    vmbblk_t *mb;

    while (vcanput(q->vq_next)) {
        mb = vgetq(q);
        if (mb == NULL) {
            if (q->vq_ptr == FLAG_STOP) {
                q->vq_ptr = FLAG_START;
                send_char(VWR(q), XON);
            }
            return 0;
        } else vputnext(q, mb);
    }
    if (q->vq_ptr == FLAG_START) {
        q->vq_ptr = FLAG_STOP;
        send_char(VWR(q), XOFF);
    }
    return 0;
}
```

Wie wird in der Empfangsroutine ein Zeichen in Treiberrichtung zurückgeschickt? Zu diesem Zweck sind immer genau zwei Queues miteinander verknüpft, so daß mit einer Reihe von Makros die richtige Queue ausgewählt werden kann. Diese Makros lauten **VWR**, **VRD**, **VWRQ**, **VRDQ** und **VOTHERQ**. In unserem Fall soll in der Empfangsroutine die Schreibqueue angesprochen werden, also benutzen wir **VWR**, **VWRQ** oder **VOTHERQ**. Unterschiede bestehen in diesem Fall nur in der Optimierung des Codes.

Das Senden eines Zeichens wird durch die folgende Funktion erledigt:

```
static int
send_char(vqueue_t *q, char c)
{
    vmbblk_t *msg = vallocb(1, 0);

    if (!msg) return -1;
    msg->vb_wptr[0] = c;
    msg->vb_wptr++;
    vputnext(q, msg);
    return 0;
}
```

1.4.5. Füllstände und -grenzen

Der Mechanismus, der hinter der oben beschriebenen Queueverwaltung steckt, benutzt zwei Marken zur Kennzeichnung der Füllstände oder -grenzen einer Queue: die high und low watermarks (**vmodule_info**, **vmi_hiwat** und **vmi_lowat**). Sobald durch ein **vputq** die obere Marke (**vmi_hiwat**) überschritten wird, liefert das Macro **vcanput** FALSE zurück und die Serviceroutine wird nicht mehr angesprochen. Die untere Marke definiert den Punkt, ab dem die Serviceroutine wieder angesprochen wird, wenn das darüberliegende Modul genügend Daten ausgelesen hat.

- Die untere Grenze (low watermark) muß immer größer als die maximale Blockgröße des Moduls sein. Hier das Beispiel für das XON/XOFF-Modul:

```
#define XONXOFF_HIGH 2048
#define XONXOFF_LOW 512

static struct _vmodule_info xonxoff_info = {
    "xonxoff",
    0,
    0,
```

```
XONXOFF_HIGH,  
XONXOFF_LOW,  
"1.1",  
V_MEDIA_PACKAGE_UTIL,  
V_MEDIA_PACKAGE_UTIL_30,  
""  
};
```

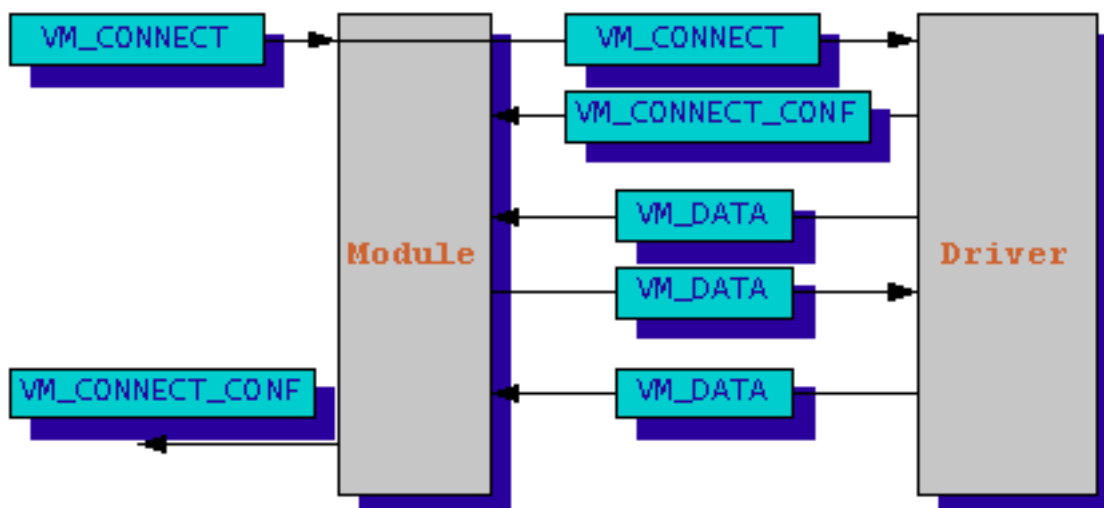
1.5. Protokollverhandlung

- 1.5.1. Einführung: Protokollverhandlung
- 1.5.2. Verbindungsaufbau im Protokollmodul
- 1.5.3. Daten von der Gegenstelle abprüfen
- 1.5.4. Daten an die Gegenstelle senden

1.5.1. Einführung: Protokollverhandlung

Das Beispielmol in diesem Kapitel soll einen typischen Protokollablauf und dessen Realisierung in der Praxis verdeutlichen. Zu den Modulen, die während des Verbindungsaufbaus eine Protokollverhandlung durchlaufen und danach Daten (fast) transparent weiterreichen, gehören z.B. **ATMODEM**, **POP3** und **TELNET**.

Module diesen Typs erhalten von der API-Schicht nach erfolgreichem Vopen-Aufruf eine **VM_CONNECT**-Message. Diese Message wird daraufhin an die darunterliegenden Protokoll- oder Treibermodule weitergereicht. Nachdem solch ein Protokollmodul von der Treiberseite eine **VM_CONNECT_CONF**-Message erhält wird das eigentliche Protokoll gestartet. Zuerst wird auf bestimmte Daten gewartet, auf die mit entsprechenden Antwortdaten reagiert wird. Sind die Protokolldaten korrekt ausgetauscht, wird die **VM_CONNECT_CONF**-Message an die API-Schicht weitergereicht und der Anwendung ein erfolgreicher Verbindungsaufbau signalisiert.



Beispieltreiber: login

Das login-Beispielmodul soll auf frei konfigurierbare Zeichenfolgen warten und darauf mit anderen, frei wählbaren Zeichenfolgen antworten. Eine Abfolge könnte z.B. so aussehen:

```
login-Modul           Gegenseite: Unix-Rechner
<---                login:
```

```

user      --->
          <--- password:
secret    --->
          <--- $          (Shell-Prompt)

```

Die Parameter für die Zeichenfolgen wurden auf 5 Parameter pro Richtung begrenzt und lauten:

EXPECT1 - erste erwartete Zeichenfolge

SEND1 - nach EXPECT1 gesendete Zeichenfolge

EXPECT2 - nach SEND1 erwartete Zeichenfolge

SEND2 - nach EXPECT2 gesendete Zeichenfolge

usw. bis

SEND5 - nach EXPECT4 gesendete Zeichenfolge

Zusätzlich gibt es den Parameter OTIMEOUT für die Zeit, nach welcher die Verbindung als gescheitert gilt, wenn auf eine EXPECTx-Zeichenfolge keine Antwort kommt.

Zur besseren Verständlichkeit sind in den folgenden Abschnitten nur Quelltextteile abgedruckt. Das komplette Modul finden Sie in der Datei login.c.

1.5.2. Verbindungsaufbau im Protokollmodul

Die wichtigste Stelle, an der Daten und Kontrollinformationen ausgewertet werden müssen, ist die rput-Funktion des Moduls:

```

static int
login_rput(vqueue_t *q, vmblock_t *mb)
{
    login_priv *p = (login_priv *)q->vq_ptr;

    switch (VDATA_TYPE(mb)) {
    case VM_CONNECT_CONF:
        vclrtimer(p->timeout);
        p->timeout = vsettimer(p->otimeout, login_timeout, (void *)q);
        vfreemsg(mb);
        return 0;
    }
}

```

An dieser Stelle wird die **VM_CONNECT_CONF**-Message herausgefiltert und ein Zeitgeber gestartet, der nach *p->otimeout* abläuft (**vsettimer**).

Jetzt ist es notwendig alle Daten mitzulesen, und zu prüfen ob die erwartete, erste Zeichenfolge auftritt:

```

case VM_DATA:
    if (p->connect == FALSE) {
        return login_check(q,mb);
    }
default:
    break;
}
vputnext(q,mb);
return 0;
}

```

Alle anderen Messages werden an das übergeordnete Modul oder die API-Schicht weitergegeben. Sollte irgendwann der Verbindungsaufbau abgeschlossen worden sein, werden auf die **VM_DATA**-Messages weitergegeben (*p->connect* ist TRUE).

1.5.3. Daten von der Gegenstelle abprüfen

Die Funktion *login_check* übernimmt das Durchsuchen der Messageblöcke nach der aktuell ausstehenden Zeichenfolge.

Das Suchen wird ähnlich wie im ersten Beispielm modul testhex.c (**Umwandeln der Daten**) mit zwei Schleifen realisiert, die alle Messageblöcke einer Message (*mb*) und alle Daten (*w*) eines Messageblocks durchlaufen.

```

static int

```

```
login_check(vqueue_t *q, vmblock_t *mb)
{
    login_priv *p = (login_priv *) q->vq_ptr;
    unsigned char *w;
    int move;

    while (mb != NULL) {
        w = mb->vb_rptr;
        while (w < mb->vb_wptr) {
```

Der folgende Quelltextausschnitt speichert ein Zeichen in einem Zwischenpuffer und kopiert 80 Zeichen der hinteren Hälfte in die vordere Hälfte, wenn der Puffer voll wurde. Dadurch sind immer genügend Zeichen für einen Vergleich vorhanden.

```
        if (p->count >= MAXBUF*2) {
            for (move = 0; move < MAXBUF; move++) {
                p->buf[move] = p->buf[MAXBUF + move];
            }
            p->count = move;
        }
        p->buf[p->count++] = *w++;
        p->buf[p->count] = '\\0';
```

Wenn noch eine Zeichenfolge erwartet wird, wird im aktuellen Puffer nach dieser gesucht. Im Erfolgsfall wird entweder eine weitere Zeichenfolge gesendet (*send_str*) oder die Verbindung ist erfolgreich aufgebaut worden. Der API-Schicht wird dies durch eine **VM_CONNECT_CONF**-Message mitgeteilt. Der restliche Messageblock wird verdoppelt (**vdupb**) und in Richtung API-Schicht weitergereicht. Das Verdoppeln ist notwendig, weil in der *rput*-Funktion unseres Beispielmotors die aktuelle **VM_DATA**-Message freigegeben wird.

```
        if (p->expect[p->expsend]) {
            if (strstr(p->buf, p->expect[p->expsend])) {
                if (p->send[p->expsend]) {
                    send_str(q);
                    p->count = 0;
                    p->expsend++;
                } else p->connect = TRUE;
            }
        }
        else p->connect = TRUE;
        if (p->connect) {
            vputctl(VRDQ(q), VM_CONNECT_CONF);
            vclrtimer(p->timeout);
            vputnext(q, vdupb(mb));
            return 0;
        }
    }
    mb = mb->vb_cont;
}
return 0;
}
```

1.5.4. Daten an die Gegenstelle senden

Diese Routine dient dazu, die aktuellen Zeichenfolge in Treiberrichtung zu senden. Damit einige Kontrollzeichen wie z.B. CR, LF, TAB usw. geschickt werden können, wird die bekannte Backslash-Methode wie in C-Strings verwendet.

Zuerst muß die Länge des resultierenden Messageblocks errechnet werden. Die Kontrollzeichenersetzung kann die Zeichenfolge höchstens verkürzen, daher können wir genau die Länge der Zeichenfolge als Datenlänge anfordern.

```
static void
```

```
send_str(vqueue_t *q)
{
    login_priv *p = (login_priv *) q->vq_ptr;
    vmblock_t *blk;
    unsigned char *pch, ch;

    blk = vallocb(strlen(p->send[p->expsend]), 0);
    if (blk == NULL) {
        vfireerror(q, VE_NO_MEMORY, "", NULL);
        p->expsend=0;
        return 0;
    }
}
```

Im folgenden Quelltextabschnitt wird die Ersetzung der Zeichen '\n', '\r', '\t', '\b' und '\ ' erledigt und der Messageblock in Treiberrichtung abgeschickt.

```
    pch = p->send[p->expsend];
    while ((ch = *pch++) != 0) {
        if (ch == '\\') {
            switch (*pch) {
                case 'r': ch = 0x0D; pch++; break;
                case 'n': ch = 0x0A; pch++; break;
                case 't': ch = 0x09; pch++; break;
                case 'b': ch = 0x08; pch++; break;
                case '\\': pch++; break;
            }
        }
        *blk->vb_wptr++ = ch;
    }
    vputnext(VWRQ(q), blk);
}
```

1.6. Treibermodule

1.6.1. Einführung: Treibermodule

1.6.2. Verbindungsaufbau im Treibermodul

1.6.3. Daten an die Anwendung senden

1.6.4. Die Leseserviceroutine des Treibers

1.6.5. Verbindungsabbau

1.6.1. Einführung: Treibermodule

Treibermodule sprechen direkt die Hardware an oder schreiben und lesen Daten durch Betriebssystemfunktionsaufrufe. LUCA-Messages werden vom LUCA-Kern durch **vallocb** angefordert und in Anwendungsrichtung im Protokollstapel nach oben weitergereicht. Von der API-Schicht kommende Datenmessages müssen von Treibermodulen erst an die Hardware oder das Betriebssystem gesendet werden, bevor sie an den LUCA-Kern mit **vfreeb** zurückgegeben werden.

Im Standardfall schickt ein Treiber alle anderen Messages, die er von *oben* bekommt, in die Gegenrichtung nach *oben* zurück (**vqreply**). Dadurch sind Treibermodule immer aufwärtskompatibel.

Beispieltreiber: testdriv

Unser Beispieltreibermodul entspricht dem **CHARGEN**-Modul, das zeilenweise Daten an die Anwendung schickt. Jeder Aufruf von Vread im blockorientierten Modus liefert eine Zeile mit ASCII-Daten. Ein einstellbarer Timer sorgt dafür, daß weitere Daten erzeugt werden, solange die Anwendung diese Daten ausliest.

In dieser Beschreibung wurden Teile des Codes zum besseren Verständnis weggelassen. Die komplette Version des Treibermoduls befindet sich in der Datei testdriv.c im Protocol Factory-Verzeichnis.

1.6.2. Verbindungsaufbau im Treibermodul

Der Verbindungsaufbau eines Treibers oder die Initialisierung der Hardware erfolgt durch eine **VM_CONNECT**-Message, die die API-Schicht bei einem **Vopen**-Aufruf schickt.

In unserem Testtreiber wird daraufhin ein Zeitgeber mit **vsettimer** initialisiert und eine **VM_CONNECT_CONF**-Message an die API-Schicht zurückgeschickt.

```
static int
testdriv_wput(vqueue_t *q, vmblock_t *mb)
{
    testdriv_priv *p = (testdriv_priv *)q->vq_ptr;

    switch (VDATA_TYPE(mb)) {
    ...
    case VM_CONNECT:
        if (p->delay) p->timer = vsettimer(p->delay, timercb, q);
        VDATA_TYPE(mb) = VM_CONNECT_CONF;
        vqreply(q, mb);
        return 0;
    ...
    default:
        break;
    }
    vqreply(q, mb);
    return 0;
}
```

Hier können Sie sehen, daß nicht unbedingt jede Message freigegeben werden muß, nur um sofort danach eine weitere Message für eine einfache Kontrollinformation anzufordern. Stattdessen kann der Messagetyp verändert werden und diese Message zurückgeschickt werden.

Die Variable *delay* im obigen Beispielcode enthält den Wert des Parameters "DELAY", der durch die Funktion *dispatchparam* auf einen Millisekundenwert gesetzt wurde.

1.6.3. Daten an die Anwendung senden

In der Funktion *timercb*, die durch den Zeitgeber aufgerufen wird, muß zuerst der Zeitgeber gelöscht werden (*vclrtimer*). Die Funktion *sendbuf* wird benutzt, wenn oberhalb des Treibermoduls Daten entgegengenommen werden können (*vcanput*).

```
static int
timercb(vqueue_t *q)
{
    testdriv_priv *p = (testdriv_priv *)q->vq_ptr;
    vclrtimer(p->timer);
    p->timer = 0;
    if (vcanput(VRDQ(q)->vq_next)) sendbuf(q);
    if (p->delay) p->timer = vsettimer(p->delay,timercb,q);
    return 0;
}
```

Die Funktion *sendbuf* füllt eine neue LUCA-Message mit Daten und schickt diese Message mit *vputnext* nach oben. Zum Schluß wird ein neuer Zeitgeber initialisiert, sollte der "DELAY"-Parameter nicht Null sein.

```
static void
sendbuf(vqueue_t *q)
{
    testdriv_priv *p = (testdriv_priv *)q->vq_ptr;
    vmbk_t *mb;
    int i;

    mb = vallocb(4+p->len+2,0);
    if (!mb) return ;
    sprintf(mb->vb_wptr,"%04d",p->lineno++);
    while (*mb->vb_wptr) mb->vb_wptr++;
    if (p->lineno > 9999) p->lineno = 0;
    for (i = 0;i < p->len; i++) {
        *mb->vb_wptr++ = *p->ps++;
        if (!*p->ps) p->ps = charset;
    }
    *mb->vb_wptr++ = '\r';
    *mb->vb_wptr++ = '\n';
    vputnext(VRDQ(q),mb);
}
```

Die Größe des Datenpuffers der Message wird aus der Zeilenlänge plus vier Zeichen für die Zeilennummer und zwei Zeichen für die Zeilenendezeichen CR und LF berechnet. Sollte nicht genügend Speicher für Datenpakete vorhanden sein, kehrt die Funktion ohne Daten zu senden zurück.

Der Datenblock, auf den *mb->vb_wptr* zeigt, wird mit der Zeilennummer, einer Zeile mit variierendem Text und den Zeilenendezeichen gefüllt und mit *vputnext* in Richtung Anwendung geschickt.

Der Zeiger *p->ps* zeigt auf ein Zeichen im Zeichensatz, der zu Beginn mit

```
static char charset[256] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 01234567890 - ";
```

initialisiert wurde.

1.6.4. Die Leseserviceroutine des Treibers

Wenn ein Treiber wieder Daten nach *oben* schicken darf, weil die Anwendung oder übergeordnete Protokolle alle Daten gelesen haben, wird vom LUCA-Schedule die Serviceroutine des Treibers aufgerufen.

```
static int
testdriv_rsvr(vqueue_t *q)
```

```

{
  testdriv_priv *p = (testdriv_priv *)q->vq_ptr;
  while (vcanput(q->vq_next) && !p->delay) {
    sendbuf(q);
  }
  if (p->delay && !p->timer) p->timer = vsettimer(p->delay,timercb,q);
  return 0;
}

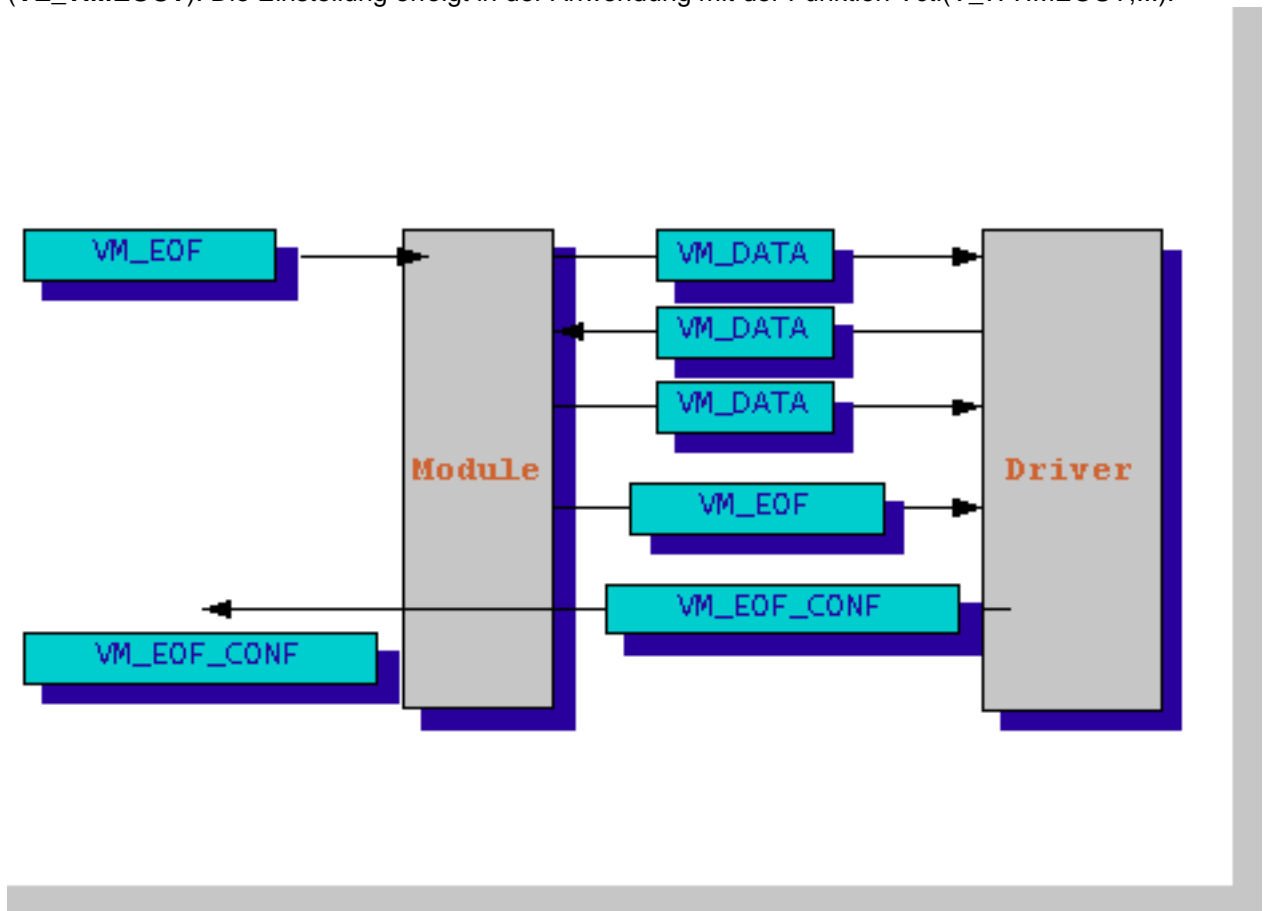
```

In unserem Beispieldriver dürfen nur Daten geschickt werden, solange übergeordnete Module Daten noch Platz in den Warteschlange zur Verfügung haben (**vcanput**). Außerdem dürfen wir keine Daten schicken, wenn der "DELAY"-Parameter auf 0 eingestellt wurde. In diesem Fall wird erst ein Zeitgeber initialisiert und das Senden entsprechend verzögert.

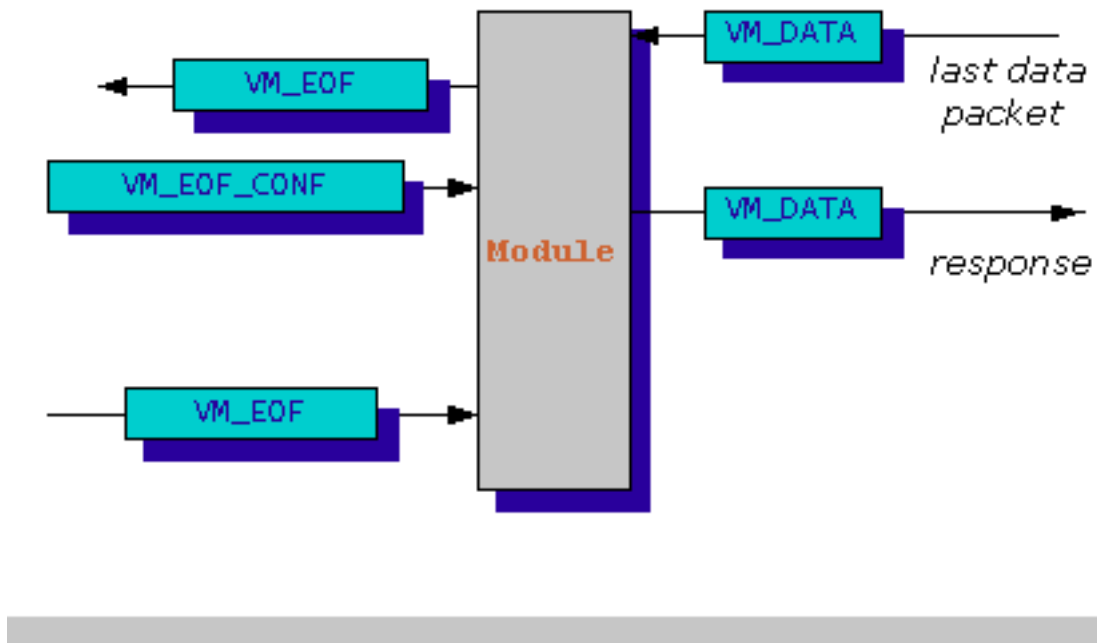
1.6.5. Verbindungsabbau

Grundsätzlich unterscheiden wir zwischen *aktivem* und *passivem* Verbindungsabbau.

- Aktiver Verbindungsabbau
Das Treibermodul erhält zum Beenden der Verbindung eine **VM_EOF**-Message von der API-Schicht. Auf diese Message muß der Treiber mit einer **VM_EOF_CONF**-Message antworten, ansonsten gilt die Verbindung als unerwartet abgebrochen und die Anwendung erhält einen Fehler (**VE_TIMEOUT**). Die Einstellung erfolgt in der Anwendung mit der Funktion `Vctl(V_WTIMEOUT,...)`.



- Passiver Verbindungsabbau
Das Treibermodul sendet zum Beenden der Verbindung selbst eine **VM_EOF**-Message an die Anwendung. Diese Message wird von der API-Schicht mit einer **VM_EOF_CONF**-Message beantwortet, sobald die Anwendung das letzte Zeichen mit `Vread()` gelesen hat. Ruft die Anwendung vor dem Lesen des letzten Zeichens ein `Vclose()` auf, gilt die Verbindung als nicht korrekt abgeschlossen. Das Treibermodul erhält daraufhin eine **VM_DATA_LOST**-Message.



Im Beispielm modul sieht dieser Code folgendermaßen aus:

```
static int
testdriv_wput(vqueue_t *q, vmbk_t *mb)
{
    testdriv_priv *p = (testdriv_priv *)q->vq_ptr;

    switch (mb->vb_datap->vdb_type) {
        ...
    case VM_EOF:
        vputctl(VRD(q), VM_EOF_CONF);
        vqreply(q, mb);
        return 0;
    case VM_EOF_CONF:
        vfreemsg(mb);
        return 0;
        ...
    }
    vqreply(q, mb);
    return 0;
}
```

Sobald der Treiber eine **VM_EOF**-Message erhält, schickt er eine **VM_EOF_CONF**-Message als Antwort zurück. Zusätzlich wird die **VM_EOF**-Message an die API-Schicht geschickt, um zu signalisieren, daß keine weiteren Daten gelesen werden können.

1.7. Spezialfunktionen

1.7.1. Protokolle unter sich

1.7.2. Automatisches Laden von Treibern

1.7.3. Schützen von Programmabschnitten

1.7.4. Der Protokollevent

1.7.1. Protokolle unter sich

In diesem Abschnitt sollen einige Spezialfunktionen erläutert werden, die der Interaktion bestimmter Module untereinander dienen.

Die beiden Funktionen **vgetproto** und **vputproto** erlauben Modulen eigene Protokollinformationen untereinander auszutauschen. Beispielsweise könnte das **ATMODEM**-Modul einen Hinweis an alle übergeordneten Module schicken, daß eine Sprachverbindung erkannt wurde. Umgekehrt könnte das **VOICE**-Modul prüfen, welche Art Medium im Protokollstack untergeordnet ist, und darauf mit bestimmten Einstellungen reagieren. Mit **vgetproto** und **vputproto** werden sozusagen Untertypen der **VM_PROTO**-Message erzeugt und verarbeitet.

VM_PROTO-Message werden auf diese Art geschickt vom ersten Modul geschickt:

```
vputnext(q, PROTO_TEST_DRIV1, "START");
```

Die Auswertung von VM_PROTO-Message im zweiten Modul sieht beispielweise so aus:

```
if (VDATA_TYPE(mb) == VM_PROTO) {
    if (vgetproto(mb, PROTO_TEST_DRIV1, NULL)) {
        ...
    } else if (vgetproto(mb, PROTO_TEST_DRIV2, NULL)) {
        ...
    } else vputnext(q, mb);
}
```

1.7.2. Automatisches Laden von Treibern

Eine weitere Funktion erlaubt das automatische Laden von zusätzlichen Modulen unterhalb des aktuellen Moduls: **vdriverattach**. Diese Funktion erwartet einen Zeiger auf die Lese-Queue des aktuellen Moduls und den Namen des Moduls, das hinzugefügt werden soll.

```
if (mode != MODULE_OPEN) {
    rc = vdriverattach(q, "socket");
    if (rc < 0) return rc;
    rc = vdriverattach(q, "tcp");
    if (rc < 0) return rc;
}
```

Sie sollten diese Funktion in der open-Funktion Ihres Moduls benutzen, wenn Sie z.B. im Link Identifier die Abkürzung "MODUL:rechner" anstelle von "MODUL/TCP:1234/SOCKET:rechner" schreiben möchten.

1.7.3. Schützen von Programmabschnitten

In manchen Fällen kann es vorkommen, daß Sie selbst Threads in Ihren Protokoll- oder Treibermodulen erzeugen. Sie müssen dabei beachten, daß selbst die kürzeste Anweisung, z.B. das Erhöhen eines Zählers, vom Betriebssystem unterbrochen werden kann. Schützen Sie Programmabschnitte, die nicht unterbrochen werden sollen mit den Funktionen **vidisable** und **viable**.

In diesem Beispiel kann der Programmabschnitt nicht unterbrochen werden:

```
int lvl;
```

```
lvl = vdisable();  
... Code kann nur durch einen Thread benutzt werden. ...  
viable(lvl);
```

Beachten Sie, daß es zu unerwarteten Nebeneffekten führen kann, wenn Sie nicht genauso oft **vdisable** wie **viable** aufrufen.

1.7.4. Der Protokollevent

Ab der LUCA-Version 2.0 steht dem Anwendungsprogrammierer ein neues Ereignis zur Verfügung, das z.Zt. nur durch das **ZMODEM**-Modul ausgelöst wird.

Das PROTOCOL-Ereignis soll dem Anwendungsprogrammierer signalisieren, daß ein weiteres Modul mit **Vpush** zum aktuellen Protokollstapel hinzugefügt werden soll.

Der Protocol Factory-Entwickler benötigt zur Kontrolle dieses Mechanismus drei Messagetypen:

- **VM_PROTCOLEV**
Diese Message wird durch die Funktion **vfireprotocol** an die API-Schicht geschickt und löst bei der Anwendung ein PROTOCOL-Ereignis aus.
- **VM_PUSHED**
Im Datenbereich von Messages dieses Typs steht der Name des Moduls, das mit Vpush über die aktuelle Verbindung gelegt wurde.
- **VM_POPPED**
Die Anwendung hat Vclose aufgerufen und für jedes Modul, das vom Protokollstapel entfernt wurde, wird dies durch solche Messages signalisiert.

2. Referenz

2.1. Messagetypen

- 2.1.1. VM_DATA
- 2.1.2. VM_EOF
- 2.1.3. VM_EXDATA
- 2.1.4. VM_ERROR
- 2.1.5. VM_PROTO
- 2.1.6. VM_SETPAR
- 2.1.7. VM_SETPAR_CONF
- 2.1.8. VM_GETPAR
- 2.1.9. VM_GETPAR_CONF
- 2.1.10. VM_CONNECT
- 2.1.11. VM_CONNECT_CONF
- 2.1.12. VM_EOF_CONF
- 2.1.13. VM_DATA_LOST
- 2.1.14. VM_ABORT
- 2.1.15. VM_NONSTDEV
- 2.1.16. VM_PROTOCOLEV
- 2.1.17. VM_PUSHED
- 2.1.18. VM_POPPED

2.2. Datentypen und Strukturen

- 2.2.1. vdblk_t
- 2.2.2. vmblok_t
- 2.2.3. vmodule_info
- 2.2.4. vmodule_stat
- 2.2.5. vparams_t
- 2.2.6. vqinit_t
- 2.2.7. vqueue_t
- 2.2.8. vstreamtab_t
- 2.2.9. vtimer_cb

2.3. Funktionsreferenz

- 2.3.1. vallocb
- 2.3.2. vfreeb
- 2.3.3. vallocq
- 2.3.4. vfreeq
- 2.3.5. vfreemsg
- 2.3.6. vdupb
- 2.3.7. vdupmsg
- 2.3.8. vcopymsg
- 2.3.9. vputctl
- 2.3.10. vputproto
- 2.3.11. vgetproto
- 2.3.12. vqenable
- 2.3.13. vputq
- 2.3.14. vputbq
- 2.3.15. vcanput
- 2.3.16. vputnext
- 2.3.17. vgetq

2.3.18. vflushq

2.3.19. vqreply

2.3.20. vfireerror

2.3.21. vfirenonstd

2.3.22. vfireprotocol

2.3.23. vdriverattach

2.3.24. vsettimer

2.3.25. vclrtimer

2.3.26. vdispatchparam

2.3.27. viable

2.3.28. vidisable

2.4. Funktionen innerhalb eines Moduls

2.4.1. module_open

2.4.2. module_close

2.4.3. module_put

2.4.4. module_service

2.1. Messagetypen

2.1.1. VM_DATA

2.1.2. VM_EOF

2.1.3. VM_EXDATA

2.1.4. VM_ERROR

2.1.5. VM_PROTO

2.1.6. VM_SETPAR

2.1.7. VM_SETPAR_CONF

2.1.8. VM_GETPAR

2.1.9. VM_GETPAR_CONF

2.1.10. VM_CONNECT

2.1.11. VM_CONNECT_CONF

2.1.12. VM_EOF_CONF

2.1.13. VM_DATA_LOST

2.1.14. VM_ABORT

2.1.15. VM_NONSTDEV

2.1.16. VM_PROTOCOLEV

2.1.17. VM_PUSHED

2.1.18. VM_POPPED

2.1.1. VM_DATA Message

Diese Message dient zum Transport von Sende- und Empfangsdaten.

Daten werden erst von der Applikation an einen Treiber geschickt, nachdem vom Treiber ein **VM_CONNECT_CONF** geschickt wurde.

1

LOW

2.1.2. VM_EOF Message

Diese Message zeigt das Ende des Datenstroms an. Die Seite, die diese Message schickt, möchte die Verbindung beenden.

2

LOW

2.1.3. VM_EXDATA Message

Z.Zt. in LUCA nicht benutzt.

1

HIGH

2.1.4. VM_ERROR Message

Diese Message wird in Richtung Anwendung geschickt, sobald ein Kommunikationsfehler die weitere Übertragung unmöglich macht. **vfireerror** sollte zum Erzeugen dieses Messagetyps verwendet werden.

2

HIGH

2.1.5. VM_PROTO Message

Mit dieser Message können Kontrollinformationen zwischen einzelnen Modulen ausgetauscht werden. **vgetproto** und **vputproto** sollten zum Erzeugen und Verarbeiten dieser Messages verwendet werden.

3

HIGH

2.1.6. VM_SETPAR Message

Diese Message wird durch einen Vsetpar-Aufruf von der Applikation erzeugt. Erhält ein Modul in der wput-Funktion diese Message, sollte **vdispatchparam** aufgerufen werden.

5

HIGH

2.1.7. VM_SETPAR_CONF Message

Diese Message wird durch **vdispatchparam** als Antwort auf eine **VM_SETPAR** Message an die Applikation geschickt.

6

HIGH

2.1.8. VM_GETPAR Message

Diese Message wird durch einen Vgetpar-Aufruf von der Applikation erzeugt. Erhält ein Modul in der wput-Funktion diese Message, sollte **vdispatchparam** aufgerufen werden.

7

HIGH

2.1.9. VM_GETPAR_CONF Message

Diese Message wird durch **vdispatchparam** als Antwort auf eine **VM_GETPAR** Message an die Applikation geschickt.

8

HIGH

2.1.10. VM_CONNECT Message

Bei Erhalt dieser Message muß ein Modul versuchen, einen Verbindungsaufbau einzuleiten.

Implementiert das Modul ein Protokoll, sollte diese Message an das darunterliegende Modul weitergegeben werden. Erst aufgrund der **VM_CONNECT_CONF** Message sollte das Protokoll soweit abgewickelt werden, bis die Verbindung steht. Dann erst muß auf ein VM_CONNECT eine **VM_CONNECT_CONF**-Message an die Applikation geschickt werden.

Handelt es sich bei dem Modul um einen Treiber, der keine Zeit für Verbindungsaufbau benötigt, kann sofort ein **VM_CONNECT_CONF** geschickt werden. Ein Protokollmodul muß in diesem Fall das **VM_CONNECT** mit vputnext weitergeben.

9

HIGH

2.1.11. VM_CONNECT_CONF Message

Diese Message beendet den Verbindungsaufbau und führt bei der Anwendung zu einem V_CONNECT-Event. Eine **VM_CONNECT_CONF**-Message darf nur als Reaktion auf eine **VM_CONNECT**-Message geschickt werden.

10

HIGH

2.1.12. VM_EOF_CONF Message

Wird diese Message von der Applikation in Treiberrichtung geschickt, ist das die Bestätigung, daß die Anwendung mit Vread(...) tatsächlich das Ende des Datenstroms korrekt gelesen hat.

Wird diese Message an die Applikation geschickt, so wird damit der korrekte Versand des letzten Datenpakets signalisiert.

11

HIGH

2.1.13. VM_DATA_LOST Message

Diese Message wird von der API-Schicht in Richtung Treiber geschickt, sobald ein Vclose-Aufruf erfolgt, obwohl noch Daten zu lesen waren. Ein Protokoll kann daraufhin der Gegenstelle signalisieren, daß nicht alle Daten korrekt bei der empfangenden Anwendung ankamen oder die Anwendung eine Verbindung abbrechen wollte.

12

HIGH

2.1.14. VM_ABORT Message

Wird diese Message vom Treiber an die Applikation geschickt, dann werden alle Daten verworfen und ein Verbindungsabbruch signalisiert. In die andere Richtung muß ein Modul (soweit möglich) der Gegenstelle signalisieren, daß die Anwendung einen Abbruch mit Vctl(V_ABORT) vorgenommen hat.

14

HIGH

2.1.15. VM_NONSTDEV Message

Diese Message wird an die Applikation geschickt und führt zu einem NONSTD-Event. Zum Erzeugen dieser Messages muß **vfirenonstd** benutzt werden.

16

HIGH

2.1.16. VM_PROTOCOLEV Message

Diese Message wird an die Applikation geschickt und führt zu einem PROTOCOL-Event. Zum Erzeugen dieser Messages muß **vfireprotocol** benutzt werden.

18

HIGH

2.1.17. VM_PUSHED Message

Diese Message wird vom LUCA-Kern an die Module geschickt, sobald die Anwendung die Funktion **Vpush** aufruft und ein entsprechendes Modul zum Protokollstapel hinzugefügt wurde. Das Gegenstück heißt **VM_POPPED**.

19

HIGH

2.1.18. VM_POPPED Message

Diese Message wird vom LUCA-Kern an die Module geschickt, sobald die Anwendung die Funktion **Vclose** aufruft und ein entsprechendes Modul vom Protokollstapel entfernt wurde. Das Gegenstück heißt **VM_PUSHED**.

20

HIGH

2.2. Datentypen und Strukturen

2.2.1. vdblk_t

2.2.2. vmblok_t

2.2.3. vmodule_info

2.2.4. vmodule_stat

2.2.5. vparams_t

2.2.6. vqinit_t

2.2.7. vqueue_t

2.2.8. vstreamtab_t

2.2.9. vtimer_cb

vdblk_t Struktur

```
typedef struct {
    struct _vblk * vdb_freep;
    vbyte * vdb_base;
    vbyte * vdb_lim;
    vui16 vdb_ref;
    vui16 vdb_type;
} vdblk_t;
```

Diese Datenblock-Struktur dient als Datenspeicher für Kontroll- und Benutzerdaten.
Ein Datablock ist immer an mindestens einen Messageblock geknüpft.

Member

vdb_freep

Wird intern von LUCA benutzt.

vdb_base

Zeigt auf den Anfang der Daten.

vdb_lim

Zeigt hinter das letzte Datenbyte.

vdb_ref

Anzahl der Messageblocks, die noch diesen Datenblock verwenden.

vdb_type

Typ dieser Message

vmblok_t Struktur

```
typedef struct {
    vmblok_t * vb_next;
    vmblok_t * vb_prev;
    vmblok_t * vb_cont;
    vbyte * vb_rptr;
    vbyte * vb_wptr;
    struct _vdatab * vb_datap;
} vmblok_t;
```

Diese Messageblock-Struktur wird verwendet, um zwischen Modulen Daten- und Kontrollinformationen auszutauschen.

Member**vb_next**

Dieser Zeiger wird intern in LUCA verwaltet. In einer Queue ist dies der Zeiger auf die nächste Message, die gelesen werden kann.

vb_prev

Dieser Zeiger wird intern in LUCA verwaltet. In einer Queue ist dies der Zeiger auf die vorhergehende Message.

vb_cont

Dieser Zeiger zeigt auf weitere Messageblöcke, die zu dieser Message gehören.

vb_rptr

Zeigt auf das erste lesbare Zeichen in dieser Message.

vb_wptr

Zeigt auf das erste beschreibbare Zeichen in dieser Message.

vb_datap

Zeigt auf den Datenblock, der die Daten dieser Message enthält.

vmodule_info Struktur

```
typedef struct {  
    vchar * vmi_name;  
    vui32 vmi_minpsz;  
    vui32 vmi_maxpsz;  
    vui32 vmi_hiwat;  
    vui32 vmi_lowat;  
    vchar * vmi_version;  
    vui32 vmi_flags;  
    vui32 vmi_flags_30;  
    const vchar * vmi_co_ver;  
} vmodule_info;
```

Diese Struktur wird zur Beschreibung von Moduleigenschaften benutzt. Es kann diese Struktur jeweils für Lese- und Schreibrichtung geben.

Member**vmi_name**

Name des Moduls

vmi_minpsz

Minimale Blockgröße, die dieses Modul akzeptiert.

vmi_maxpsz

Maximale Blockgröße, die dieses Modul akzeptiert.

vmi_hiwat

High-Water-Mark. Gibt an, bis zu welcher Anzahl Zeichen noch in diese Queue geschrieben werden darf.

vmi_lowat

Low-Water-Mark. Gibt an, ab welcher Zeichenanzahl wieder in diese Queue geschrieben werden darf.

vmi_version

Dieser String gibt die LUCA-Version an, mit der dieses Modul implementiert wurde. Wird eine ältere LUCA-Version mit diesem Modul benutzt, erscheint als Hinweis das Evaluation Fenster.

vmi_flags

Hier sind die verschiedenen LUCA-Packages bitweise codiert. V_MEDIA_PACKAGE_UTIL ist ein sinnvoller Wert, da das UTIL Package bei allen LUCA-Versionen vorhanden ist.

vmi_flags_30

Hier sind die verschiedenen LUCA-Packages ab der Version 3.0 bitweise codiert. V_MEDIA_PACKAGE_UTIL_30 ist ein sinnvoller Wert, da das UTIL Package bei allen LUCA-Versionen vorhanden ist.

vmi_co_ver

Reserviert.

vmodule_stat Struktur

```
typedef struct {
    vui32 vms_pcmt;
    vui32 vms_scmt;
    vui32 vms_ocmt;
    vui32 vms_ccmt;
} vmodule_stat;
```

Diese Struktur enthält Statistikdaten, die z.Zt. in LUCA nicht benutzt oder ausgefüllt werden.

Member**vms_pcmt**

Anzahl der Modul-put Aufrufe.

vms_scmt

Anzahl der Modul-service Aufrufe.

vms_ocmt

Anzahl der Modul-open Aufrufe.

vms_ccmt

Anzahl der Modul-close Aufrufe.

vparams_t Struktur

```
typedef struct {
    vchar * param;
    vbyte * (*) (vqueue_t *, vchar *, vsi32) func;
} vparams_t;
```

Mit diesem Tabellentyp werden Parameternamen entsprechenden Funktionen zugeordnet. Die Verwaltung und den Aufruf der Funktionen übernimmt **vdispatchparam**.

Member**param**

Zeiger auf den Parameternamen.

func

Zeiger auf eine Parameterfunktion.

vqinit_t Struktur

```
typedef struct {
    int (*) (vqueue_t *q, vmbk_t *m) vqi_putp;
    int (*) (vqueue_t *q) vqi_srvp;
    int (*) (vqueue_t *q, int omode) vqi_qopen;
    int (*) (vqueue_t *q) vqi_qclose;
    struct vmodule_info * vqi_minfo;
```

```
    struct vmodule_stat * vqi_mstat;  
} vqinit_t;
```

Struktur für Modul-Initialisierungsroutinen. Diese Struktur enthält Zeiger auf die jeweiligen put-, service-, open- und close-Funktionen für die Schreib- und Leserichtung.

Member

vqi_putp

Zeiger auf die **module_put** Funktion.

vqi_srvp

Zeiger auf die **module_service** Funktion.

vqi_qopen

Zeiger auf die **module_open** Funktion.

vqi_qclose

Zeiger auf die **module_close** Funktion.

vqi_minfo

Zeiger auf die Modulinformationsdaten.

vqi_mstat

Zeiger auf die Modulstatistik.

vqueue_t Struktur

```
typedef struct {  
    vqinit_t * vq_qinfo;  
    vmblk_t * vq_first;  
    vmblk_t * vq_last;  
    vqueue_t * vq_next;  
    vqueue_t * vq_link;  
    void * vq_ptr;  
    vui32 vq_count;  
    vui32 vq_flag;  
    vui32 vq_minpsz;  
    vui32 vq_maxpsz;  
    vui32 vq_hiwat;  
    vui32 vq_lowat;  
} vqueue_t;
```

Eine Queue gibt es immer in Lese- und in Schreibrichtung.

Member

vq_qinfo

Zeiger auf die Modulinitialisierungstabelle.

vq_first

Reserviert.

vq_last

Reserviert.

vq_next

Reserviert.

vq_link

Reserviert.

vq_ptr

Dieser Zeiger kann für modulinterne Zwecke belegt werden. Der Speicher sollte in der Modul-open Funktion allokiert und in der Modul-close Funktion freigegeben werden.

vq_count

Reserviert.

vq_flag

Reserviert.

vq_minpsz

Kopie der Modulinformation.

vq_maxpsz

Kopie der Modulinformation.

vq_hiwat

Kopie der Modulinformation.

vq_lowat

Kopie der Modulinformation.

vstreamtab_t Struktur

```
typedef struct {  
    vqinit_t * vst_rdinit;  
    vqinit_t * vst_wrinit;  
    vqinit_t * vst_muxrinit;  
    vqinit_t * vst_muxwinit;  
} vstreamtab_t;
```

Diese Tabelle faßt die Datenstrukturen für die Lese- und Schreibrichtung zusammen und bildet die zentrale Adresse, die das Modul an den LUCA-Kern übergibt. Außer dieser Tabelle sollten keine weiteren Daten in die Objektdatei exportiert werden, damit es zu keinen Namenskonflikten mit anderen Modulen kommt.

Member**vst_rdinit**

Zeiger auf die Modulinformationen für die Leserichtung.

vst_wrinit

Zeiger auf die Modulinformationen für die Schreibrichtung.

vst_muxrinit

Zeiger auf die Modulinformationen für die Multiplexer-Leserichtung. Z.Zt. von LUCA unbenutzt.

vst_muxwinit

Zeiger auf die Modulinformationen für die Multiplexer-Schreibrichtung. Z.Zt. von LUCA unbenutzt.

vtm

Dieser Datentyp wird von **vsettimer** und **vcrltimer** benutzt. Es handelt sich um eine interne, fortlaufende Nummer, die jeden Timer eindeutig auswählt.

vtimer_cb

void vtimer_cb(void * ptr)

Callbackfunktionen für Timeouts. Dieser Typ von Funktionen wird für **vsettimer** benutzt.

Parameter

ptr

Zeiger auf Benutzerdaten.

Siehe auch

vsettimer **vcrltimer**

2.3. Funktionsreferenz

- 2.3.1. `vallocb`
- 2.3.2. `vfreeb`
- 2.3.3. `vallocq`
- 2.3.4. `vfreeq`
- 2.3.5. `vfreemsg`
- 2.3.6. `vdupb`
- 2.3.7. `vdupmsg`
- 2.3.8. `vcopymsg`
- 2.3.9. `vputctl`
- 2.3.10. `vputproto`
- 2.3.11. `vgetproto`
- 2.3.12. `vqenable`
- 2.3.13. `vputq`
- 2.3.14. `vputbq`
- 2.3.15. `vcanput`
- 2.3.16. `vputnext`
- 2.3.17. `vgetq`
- 2.3.18. `vflushq`
- 2.3.19. `vqreply`
- 2.3.20. `vfireerror`
- 2.3.21. `vfirenonstd`
- 2.3.22. `vfireprotocol`
- 2.3.23. `vdriverattach`
- 2.3.24. `vsettimer`
- 2.3.25. `vclrtimer`
- 2.3.26. `vdispatchparam`
- 2.3.27. `vienable`
- 2.3.28. `vidisable`

vallocb

`vmbk_t * vallocb(int size, int prio)`

Diese Funktion allokiert einen Messageblock.

Der Messageblock muß mit **vfreeb** wieder freigegeben werden oder mit einer der put-Funktionen **vputnext**, **vqreply** usw. weitergegeben werden.

Der Typ der Message wird beim Allokieren automatisch auf **VM_DATA** gesetzt. Die Zeiger `vb_wptr` und `vb_rptr` in der Message, zeigen auf den Anfang des Datenbereichs des Datenblocks (**vdblk_t**, `vdb_base`).

Rückgabewert

Gibt einen der folgenden Werte zurück:

NULL

Wenn nicht mehr genügend Speicher vorhanden ist.

!= NULL

Einen Zeiger auf eine **vmbk_t** Struktur.

Parameter

size

Anzahl der Bytes, die für den Datenteil der Message reserviert werden sollen.

prio

Dieser Parameter ist immer auf 0 zu setzen.

vfreeb

void vfreeb(vmbk_t * msgb)

Diese Funktion gibt einen Messageblock frei.

Um eine komplette Message, die aus mehreren Messageblöcken bestehen kann, freizugeben, sollte **vfreemsg** aufgerufen werden.

Parameter

msgb

Zeiger auf einen Messageblock

vallocq

vqueue_t * vallocq(void)

Diese Funktion allokiert Speicher für zwei Queue-Strukturen, also jeweils die Lese- und Schreibrichtung.

Zum Freigeben muß die Funktion **vfreeq** benutzt werden.

Diese Funktion wird normalerweise nur vom LUCA-Kern verwendet.

Rückgabewert

Gibt einen der folgenden Werte zurück:

NULL

wenn nicht genügend Speicher vorhanden war

!= NULL

Einen Zeiger auf die Lese-Queue

vfreeq

void vfreeq(vqueue_t * queue)

Diese Funktion gibt eine Queue-Struktur frei, die zuvor mit **vallocq** angefordert wurde.

Diese Funktion wird normalerweise nur vom LUCA-Kern verwendet.

Parameter

queue

Zeiger auf die Queue, die freigegeben werden soll.

vfreemsg

void vfreemsg(vmblok_t * msg)

Diese Funktion gibt eine Message mit allen Messageblöcken frei.

Parameter

msg

Zeiger auf die Message, die freigegeben werden soll

vdupb

vmblok_t * vdupb(vmblok_t * msgb)

Diese Funktion dupliziert einen einzelnen Messageblock.

Im Gegensatz zum Kopieren mit **vcopyb** ist diese Funktion effektiver, da nur ein Referenzzähler erhöht wird.

Der Messageblock muß mit **vfreeb** wieder freigegeben werden oder mit einer der put-Funktionen **vputnext**, **vqreply** usw. weitergegeben werden.

Rückgabewert

Gibt einen der folgenden Werte zurück:

NULL

wenn kein Speicher für diese Operation vorhanden ist

!= NULL

einen Zeiger auf einen neuen Messageblock

Parameter

msgb

Zeiger auf einen Messageblock, der dupliziert werden soll.

vdupmsg

vmblok_t * vdupmsg(vmblok_t * m)

Diese Funktion dupliziert eine komplette Message mit allen Datenblöcken, die zu der Message gehören.

Im Gegensatz zu **vcopymsg** werden mit **vdupmsg** keine Daten kopiert, sondern nur die Anzahl der Referenzen auf die Datenblöcke erhöht.

Zum Duplizieren eines einzelnen Datenblocks sollte **vdupb** benutzt werden.

Die Message muß mit **vfreemsg** wieder freigegeben werden oder mit einer der put-Funktionen **vputnext**, **vqreply** usw. weitergegeben werden.

Rückgabewert

Gibt einen der folgenden Werte zurück:

NULL

wenn nicht genügend Speicher vorhanden war

!= NULL

Zeiger auf eine neue Message

Parameter

m

Zeiger auf die zu duplizierende Message

vcopymsg

vmbk_t * vcopymsg(vmbk_t * msg)

Diese Funktion kopiert eine komplette Message, d.h. alle Datenblöcke, die zu der Message gehören.

Zum Kopieren eines einzelnen Datenblocks sollte **vcopyb** benutzt werden.

Die Message muß mit **vfreemsg** wieder freigegeben werden oder mit einer der put-Funktionen **vputnext**, **vqreply** usw. weitergegeben werden.

Rückgabewert

Gibt einen der folgenden Werte zurück:

NULL

wenn nicht genügend Speicher vorhanden war

!= NULL

Zeiger auf eine neue Message

Parameter

msg

Zeiger auf die zu kopierende Message

vputctl

void vputctl(vqueue_t * queue, unsigned short type)

Diese Funktion legt eine einzelne Message, die keine Daten enthält, in der Queue ab. Damit werden meist Kontrollinformationen ausgetauscht.

Parameter

queue

Zeiger auf die Queue, in der die Kontrollmessage abgelegt werden soll.

type

Typ der Kontrollmessage

Beispiel

Senden von **VM_CONNECT_CONF** an das nächste Modul:

```
vputctl(q->vq_next, VM_CONNECT_CONF);
```

vputproto

void vputproto(vqueue_t * queue, unsigned short proto, char * str)

Diese Funktion erzeugt eine **VM_PROTO** Message, die zum Austausch von modulspezifischen Protokollinformationen genutzt wird.

vputproto schickt diese Message mit **vputnext** an das nächste Modul in der Queue.

Parameter

queue

Zeiger auf die Queue, an die Protokollinformationen geschickt werden sollen

proto

Protokoll-ID.

str

Zeiger auf die Protokollinformationen. Hierbei handelt es sich um einen nullterminierten String.

Beispiel

Senden von Protokollinformationen:

```
vputproto(q, PROTO_TEST_DRIV, "START");
```

Siehe auch

VM_PROTO

vgetproto

vgetproto

int vgetproto(vmbk_t * msg, unsigned short proto, char * str)

Diese Funktion analysiert **VM_PROTO**-Messages und vergleicht den Inhalt mit der Protokoll-ID *proto*.

Stimmt der *proto* mit dem Wert in der Message *msg* überein, wird der String nach *str* kopiert und die Message freigegeben.

Gibt es keine Übereinstimmung wird, die Message nicht freigegeben. Nach erfolglosen Tests der Message mit *vgetproto* **muß** die Message weitergegeben werden.

Rückgabewert

Gibt einen der folgenden Werte zurück:

0

Die Protokoll-ID *proto* stimmt nicht überein.

!= 0

Die Protokoll-ID *proto* stimmt überein und die Message wurde freigegeben.

Parameter

msg

Zeiger auf eine Message

proto

Protokoll-ID

str

Zeiger auf einen Speicherbereich, in den der String kopiert werden kann.

Die Länge des Strings entspricht der Länge von *msg* -2.

Wenn *str* NULL ist, werden keine Zeichen des Strings kopiert.

Beispiel

Auswerten von Protokollinformationen:

```
if (VDATA_TYPE(mb) == VM_PROTO) {
    if (vgetproto(mb, PROTO_TEST_DRIV1, NULL)) {
        ...
    } else if (vgetproto(mb, PROTO_TEST_DRIV2, NULL)) {
        ...
    } else vputnext(q, mb);
}
```

Siehe auch

VM_PROTO

vputproto

vqenable

void vqenable(vqueue_t * queue)

Mit dieser Funktion wird ein **module_service** Routinenaufwurf eingeleitet. Der Aufruf erfolgt ggf. zu einem späteren Zeitpunkt, je nach dem, welche anderen Module noch Daten verarbeiten wollen.

venableok und **vnoenable** sind Funktionen, die die Aufrufe der **module_service** Routine zusätzlich steuern.

Parameter

queue

Zeiger auf die Queue, die wieder Daten verarbeiten kann.

vputq

int vputq(vqueue_t * queue, vmbk_t * msg)

Diese Funktion legt eine Message am Ende der Queue ab. Die Message wird mit **vgetq** nach allen anderen Messages in der Queue gelesen.

Rückgabewert

vputq liefert 0 zurück.

Parameter

queue

Zeiger auf die Queue, in der die Message abgelegt werden soll.

msg

Zeiger auf die Message, die abgelegt werden soll

vputbq

void vputbq(vqueue_t * queue, vmbk_t * msg)

Mit dieser Funktion wird eine Message an den Anfang einer Queue zurückgelegt. Diese Message kann später mit **vgetq** wieder aus der Queue herausgeholt werden.

Parameter

queue

Zeiger auf die Queue, in die die Message zurückgelegt werden soll.

msg

Zeiger auf die Message, die zurückgelegt werden soll.

vcanput

int vcanput(vqueue_t * queue)

Diese Funktion testet ob in die Queue geschrieben werden kann. Im Normalfall wird damit in einem Modul geprüft, ob die nächste Queue beschreibbar ist.

Rückgabewert

vcanput liefert den Zustand der *queue*

0
wenn die Queue nicht beschreibbar ist
!= 0
wenn die Queue beschreibbar ist

Parameter

queue
Ein Zeiger auf eine Queue

Beispiel

Das Beispiel zeigt eine typische Schleife zur Übernahme von Daten aus einer Queue in die nächste Queue

```
while (vcanput(q->vq_next)) {  
    if ((mb = vgetq(q)) != NULL)  
        vputnext(q, mb);  
    else  
        break;  
}
```

vputnext

void vputnext(vqueue_t * queue, vmbk_t * msg)

Mit diesem Makro wird die Routine **module_put** des nächsten Moduls aufgerufen. Damit werden Messages von einem Modul zum nächsten weitergegeben.

Parameter

queue
Zeiger auf die eigene Queue

msg
Zeiger auf die Message, die an das nächste Modul weitergegeben werden soll

vgetq

vmbk_t * vgetq(vqueue_t * queue)

Diese Funktion liefert die erste Message der Queue.

Die Message muß mit **vfreemsg** wieder freigegeben werden oder mit einer der put-Funktionen **vputnext**, **vqreply** usw. weitergegeben werden.

Rückgabewert

vgetq liefert im Erfolgsfall einen Zeiger auf eine Message.

NULL
Es ist keine Message vorhanden

!= NULL
Einen Zeiger auf eine **vmbk_t** Struktur.

Parameter

queue
Zeiger auf die Queue, von der die erste Message geliefert werden soll.

vflushq

void vflushq(vqueue_t * queue, int which)

Diese Funktion gibt alle Messages in einer Queue frei.

Parameter

queue

Ein Zeiger auf die Queue, die gelöscht werden soll

which

Hier kann entweder VFLUSHALL oder VFLUSHDATA angegeben werden.

VFLUSHALL löscht alle Messages der Queue.

VFLUSHDATA löscht nur die Messages vom Typ **VM_DATA** und **VM_EXDATA**

vqreply

void vqreply(vqueue_t * q, vmbk_t * msg)

Diese Funktion schickt eine Message in die andere Richtung zurück. Dies entspricht einem Aufruf von vputnext(VOTHERQ(q),mb).

vqreply sollte zum Beantworten von Kontrollinformationen genutzt werden. Wenn es sich um eine Message handelt, die beantwortet wird, kann mit **VDATA_TYPE** der Typ umgesetzt werden und die Antwort zurückgeschickt werden, ohne daß eine neue Message allokiert werden muß.

Parameter

q

Zeiger auf die eigene Queue.

msg

Zeiger auf die Message, die zurückgeschickt werden soll.

Beispiel

Beantworten von Kontrollinformationen

```
if (VDATA_TYPE(mb) == VM_CONNECT) {
    VDATA_TYPE(mb) = VM_CONNECT_CONF;
    vqreply(q,mb);
}
```

Siehe auch

vputnext

vfireerror

void vfireerror(vqueue_t * queue, int ecode, char * helpstr, vmbk_t * msg)

Diese Funktion löst bei der Anwendung einen Fehler aus und führt zu einem ERROR-Event, wenn der Vopen() Aufruf bereits abgeschlossen ist.

Vread() und Vwrite() Aufrufe sind daraufhin nicht mehr möglich und die Anwendung erhält den Fehler *ecode*.

Parameter

queue

Die Queue des aktuellen Moduls.

ecode

Die Fehlernummer, die die Anwendung erhält

helpstr

Ein Zeiger auf einen modulspezifischen Hilfstext, der den Fehler beschreibt oder NULL.

msg

Die Message, die den Fehler auslöste oder NULL. Wenn ein Zeiger auf eine Message übergeben wird, wird die Message von LUCA freigegeben.

Beispiel

Dieses Beispiel zeigt eine typische Fehlerbehandlung bei **VM_CONNECT**

```
switch (V_DATA_TYPE(msg)) {
  case VM_CONNECT:
    if (p->param == 0) {
      vfireerror(q, VE_IO_FAIL, "PARAM not set", msg);
    } else {
      ...
    }
  ...
}
```

Siehe auch

vfireonstd

vfireprotocol

VM_ERROR

vfireonstd

void vfireonstd(vqueue_t * queue, char * event)

Diese Funktion löst bei der Anwendung einen NONSTD-Event aus.

LUCA sorgt dafür, daß nicht zu viele NONSTD-Events durch Module ausgelöst werden können, die von der Anwendung nicht mit `Vgetpar("NONSTD",...)` ausgelesen werden.

Parameter

queue

Die Queue des aktuellen Moduls.

event

Ein Zeiger auf einen modulspezifischen Eventnamen

Der Eventname darf keine Punkte enthalten und sollte nur aus Buchstaben und Ziffern bestehen!

Der Modulname, den die Anwendung beim Aufruf von `Vgetpar("NONSTD",...)` mit ausliest, wird von **vfireonstd** automatisch ergänzt.

Beispiel

Auslösen eines modulspezifischen Events

```
vfireonstd(q, "PANIC");
```

Siehe auch

vfireprotocol

vfireerror

VM_NONSTDEV

vfireprotocol

void vfireprotocol(vqueue_t * queue, char * event)

Diese Funktion löst bei der Anwendung einen PROTOCOL-Event aus.

Parameter

queue

Die Queue des aktuellen Moduls.

event

Ein Zeiger auf den Namen eines anderen Moduls.

Der Protokollname darf keine Punkte enthalten und sollte nur aus Buchstaben und Ziffern bestehen!
Der Modulname, den die Anwendung beim Aufruf von Vgetpar("PROTOCOL",...) mit ausliest, wird von **vfireprotocol** automatisch ergänzt.

Beispiel

Auslösen einer Protokollumschaltung

```
vfireprotocol(q, "ZMODEM");
```

Siehe auch

vfirenonstd

vfireerror

VM_PROTOCOLEV

vdriverattach

int vdriverattach(vqueue_t * queue)

Mit dieser Funktion kann ein weiteres Protokollmodul oder ein Treiber an das aktuelle Modul angebunden werden.

Rückgabewert

Liefert den Rückgabewert der Funktion `module_open` des angebundenen Moduls

0

Das neue Modul wurde in den Protokollstack eingebaut.

!= 0

Ein LUCA Fehlercode. (s. **Fehlercodes**)

Parameter

queue

Die Queue, an der das Modul angebunden werden soll. Dies ist im Normalfall die gleiche Queue, die auch beim Öffnen in **module_open** angegeben wurde.

vsettimer

vtm vsettimer(int timeout, vtimer_cb func, void * ptr)

Mit dieser Funktion wird ein plattformunabhängiger Timer erzeugt. Wenn das Timeout abläuft, wird die Callbackfunktion `func` aufgerufen. Der Timer muß daraufhin mit **vcrltimer** gelöscht werden.

Rückgabewert

vsettimer liefert eine systemabhängige, eindeutige Nummer für diesen Timer zurück. Wenn 0 zurückgeliefert wird, konnte kein weiterer Timer eingerichtet werden.

Parameter*timeout*

Wartezeit in Millisekunden.

func

Zeiger auf die Funktion, die nach Ablauf des Timeouts aufgerufen werden soll.

ptr

Zeiger auf eine Benutzerdatenstruktur. Dieser Zeiger kann frei verwendet werden und wird in **vtimer_cb** an das Modul übergeben.

Beispiel

Verwendung von **vsettimer** und **vclrtimer**

```
static int
testdriv_timer(void *p)
{
    vqueue_t *q = (vqueue_t *)p;
    testdriv_priv *priv = (testdriv_priv *)q->vq_ptr;
    vfireonstd(q, "ERR_TIMEOUT");
    vclrtimer(priv->timer);
    return 0;
}
...
priv->timer = vsettimer(1200, testdriv_timer, q);
```

vclrtimer**void vclrtimer(vtm timer)**

Mit dieser Funktion wird ein Timer freigegeben.

Auf jeden Aufruf von **vsettimer** muß immer ein Aufruf von **vclrtimer** erfolgen.

Parameter*timer*

Von **vsettimer** zurückgelieferter Timer.

vdispatchparam**void vdispatchparam(vqueue_t * q, vmbk_t * msg, vparams_t * param_table)**

Diese Funktion vergleicht die angegebene Message mit den Tabelleneinträgen in *param_table* und ruft ggf. die dem Parameter entsprechende Funktion auf.

Die Message *msg* wird durch **vdispatchparam** freigegeben.

Parameter*q*

Zeiger auf die Queue, von der die Message stammt.

msg

Zeiger auf die Message, die eine **VM_SETPAR** oder **VM_GETPAR** Message enthält.

param_table

Zeiger auf die Parametertabelle.

vienable

void vienable(int level)

Diese Funktion erlaubt wieder Mehrfachzugriff, der zuvor mit **vidisable** verhindert wurde.

Parameter

level

Wert, der von **vidisable** geliefert wurde.

Beispiel

Schützen von bestimmten Codeabschnitten gegen Mehrfachzugriff:

```
int lvl;  
  
lvl = vidisable();  
... Code kann nur durch einen Thread benutzt werden. ...  
vienable(lvl);
```

Siehe auch

vidisable

vidisable

int vidisable(void)

Diese Funktion verhindert mehrfachen Zugriff auf gemeinsame Datenstrukturen innerhalb eines Moduls.

Rückgabewert

vidisable gibt einen Wert zurück, der bei **vienable** angegeben werden muß.

Beispiel

Schützen von bestimmten Codeabschnitten gegen Mehrfachzugriff:

```
int lvl;  
  
lvl = vidisable();  
... Code kann nur durch einen Thread benutzt werden. ...  
vienable(lvl);
```

Siehe auch

vienable

2.4. Funktionen innerhalb eines Moduls

2.4.1. module_open

2.4.2. module_close

2.4.3. module_put

2.4.4. module_service

module_open

int module_open(vqueue_t * queue, int open_mode)

Wird beim Öffnen eines Moduls aufgerufen

Parameter

queue

Zeiger auf die Lese-Queue des Moduls

open_mode

Der Modus, mit dem das Modul geöffnet wird.

Wird das Modul als Treiber geöffnet ist *open_mode* VDRIVER_OPEN sonst VMODULE_OPEN.

module_close

int module_close(vqueue_t * queue)

Wird beim Schließen eines Moduls aufgerufen

Parameter

queue

Zeiger auf die Lese-Queue Moduls

module_put

int module_put(vqueue_t * queue, vmblock_t * msg)

Wird bei der Übergabe von Messages durch z.B. **vputnext** aufgerufen.

Parameter

queue

Zeiger auf die entsprechende Queue des Moduls

msg

Zeiger auf die Message, die verarbeitet werden soll

module_service

int module_service(vqueue_t * queue)

Wird durch den LUCA-Scheduler aufgerufen. Diese Funktion muß dafür sorgen, daß solange Daten von der eigenen Queue an die nächste Queue übergeben werden, bis entweder die eigene Queue leer ist, oder die nächste Queue keine weiteren Daten erhalten darf (siehe **vcanput**).

Parameter

queue

Zeiger auf die entsprechende Queue des Moduls