CMS 3 - Handbuch

Autoren: Frank Reglin, S.Kortmann, Condat GmbH

Dokumentenversion: 0.4

Inhalt

Teil 1 : Allgemeines

1. Was ist CMS?

Technische Systeme müssen in der Lage sein, schnell und quasi gleichzeitig auf Anforderungen der Außenwelt zu reagieren. Software für solche Systeme muß diese Notwendigkeit berücksichtigen. Die beste Grundlage dafür ist ein Multitasking Betriebssystem.

Software für technische Systeme muß wechselnden Hardware-Anforderungen gerecht werden, um Investitionen zu schützen, skalierbare Einsatzfälle zu entwickeln oder auch um den Unterschied zwischen der Entwicklungsplattform und der Zielplattform gering zu halten. Ein portables System ist dafür die Lösung.

CMS (Condat Multitaskingsystem) ist ein Multitasking Betriebssystem für den Einsatz auf verschiedenen Hardware-Plattformen. Es läuft auf den gängigsten CPUs, z.B. solchen mit Intel-Architektur oder M68332 von Motorola.

CMS unterstützt die Zerlegung einer Applikation in mehrere voneinander unabhänge, miteinander kommunizierenden Prozesse und deren quasiparallele Abarbeitung. Es verwaltet zu diesem Zweck Prozesse, Timer, Semphoren, Events, Queues, Pipes und Driver. Sie werden im folgenden Text als "CMS-Objekte" oder einfach "Objekte" näher erläutert.

Die Applikationen nutzen die Dienste von CMS über eine C-Schnittstelle, die auf allen Plattformen gleich ist. Die Schnittstellen-Funktionen von CMS sind Gegenstand des Referenzteils dieses Handbuches.

Hier zunächst einmal die Highlights von CMS im Überblick:

- Auf mehreren Plattformen vorhanden mit derselben C-Schnittstelle
- Modulare, dem Anwendungsfall entsprechende Konfigurationsmöglichkeiten
- Wahlweise preemptives oder kooperatives Multitasking
- Dynamische Erzeugung und Entfernung aller CMS-Objekte zur Laufzeit
- Statische und dynamisch veränderbare Pozeßprioritäten
- Inter-Prozeß-Kommunikation über Message-Queues oder Pipes
- Transparentes Umlenken der Inter-Prozeß-Kommunikation mit Filterungsmöglichkeit
- Übertragung der Daten wahlweise als Kopie oder als Referenz
- Priorisierte Nachrichten
- Umfangreiche und effiziente Signalisierungsmöglichkeiten
- Synchronisation über Semaphoren und Events
- Komfortable Timer-Funktionalität
- Aufruf von Timeout-Funktionen im Prozeß-Kontext
- Dynamisches Laden von Drivern zur Laufzeit
- Effizientes eigenes Speichermanagement

1.1. Eigenschaften von CMS

Portabilität

CMS bietet auf allen Plattformen dieselbe C-Schnittstelle an. Systembedingte Unterschiede werden durch spezielle CMS-Typen oder Makros verdeckt. Die damit erlangte Source-Kompatiblität ermöglicht den Austausch aller nicht hardware-abhängigen Software-Komponenten zwischen den verschiedenen Hard- und Software-Plattformen ohne Source-Code-Änderungen. Zur Übernahme einer Software-Komponente ist nur eine Kompilierung erforderlich.

Konfigurierbarkeit und Erweiterbarkeit

Einzelne Teile von CMS, die im konkreten Fall nicht benötigt werden, lassen sich beim Linken ausblenden. CMS kann reduziert werden bis auf einen µ-Kernel, der immer vorhanden sein muß.

Die Funktionen für die Inter Process Communication (IPC) sind für alle IPC-Objekte (Queue, Pipe, Communication Driver) einheitlich parametrisiert. Damit sind die Mechanismen leicht austauschbar.

Mittels Treibern (Schnittstellen-Treiber und Funktionsbibliotheken) kann die Funktionalität von CMS erweitert werden.

Betriebsarten

CMS unterstützt durch unterschiedliche Betriebsarten sowohl das "preemptive multitasking" (unterbrechend, mit time slicing) als auch das "non-preemptive multitasking" (kooperativ, ohne time slicing).

CMS ist ein stand alone System, d.h. es setzt direkt auf dem Prozessor auf .

Nachladbare Objekte

Alle Objekte (Prozesse, Driver, Queues, Timer, ...) können zur Laufzeit erzeugt und auch wieder aus dem System entfernt werden. Programme können unabhängig vom Betriebssystem nachgeladen werden. Damit können bestehende Komponenten gegen weiterentwickelte Spezialversionen ausgetauscht werden.

Testunterstützung

Eine der Stärken von CMS ist die Unterstützung beim Test des Zusammenspiels der Prozesse. CMS hat einen zentralen Mechanismus zur Umlenkung (Redirection) der transportierten Daten. Damit kann man Daten, die zwischen den Prozessen oder auch mit Communication Drivern ausgetauscht werden, zu einer anderen Zielinstanz umlenken, man kann die Kommunikation abhören oder manipulieren.

Die Daten können dabei von und zu Queues, Pipes und einer bestimmte Klasse von Drivern (Communication Driver) umgelenkt werden, allgemeiner für alle Objekte, die die Standard-Funktionen zur Inter-Prozeß-Kommunikation anbieten. Die Umlenkung kann dabei auch von einem Objekttyp zu einem anderen erfolgen.

1.2. Konventionen

CMS stellt den Anwendungen Funktionen zum Zugriff auf die Objekttypen Prozeß, Timer, Semaphore, Event, Queue, Pipe und Driver zur Verfügung.

Jedes Objekt in CMS besitzt einen Namen und eine Nummer (Handle). Die Handles werden von CMS vergeben; der erste Zugriff erfolgt daher immer über den Namen.

Sowohl Handle als auch Name sind innerhalb der Objekttypen eindeutig. Es kann aber Objekte verschiedener Typen geben, die denselben Namen haben. Zum Beispiel könnte der Prozeß 'XYZ' eine Queue 'XYZ' erzeugen.

Für alle CMS-Funktionen und ihre Darstellung in diesem Handbuch gelten die folgenden Konventionen:

- Der Funktionsname hat immer den Aufbau y_ffff. Dabei steht y für einen Objekttyp oder für eine Funktionengruppe:

  p = Prozeß t = Timer r = Semaphore
  e = Event q = Queue m = Pipe (Mailbox)
  d = Driver u = Utility-Funktionen s = allgemeine System-Funktionen,

  ffff kennzeichnet die Funktionalität.

- Alle Funktionen haben einen Returncode. Dieser ist fast immer vom Typ CMS_RETURN, in einigen Fällen CMS_HANDLE oder “int” und repräsentiert ein Handle (> 0), einen Errorcode (< 0), eine erfolgreiche Durchführung der Funktion (= 0) oder eine Byteanzahl. Die Errorcodes sind für die CMS-Funktionen einheitlich definiert.

- Datentypen, für die eine bestimmte Länge vorausgesetzt wird, werden mit CMS-weit gültigen Typdeklarationen vereinbart. Z.B. soll ein Signalmaske immer 32 Bit lang sein und erhält den Typ CMS_SIGNAL.

Die allgemeinen Typdeklarationen und Errorcodes befinden sich in CMS.H, die prozessorspezifischen in CMS_TYPES.H.

1.3. Standard-Funktionen auf CMS-Objekten

Für alle von CMS verwalteten Objekte wird ein Satz von Management-Funktionen angeboten, die für die Verwaltung der Objekte der einzelnen Typen erforderlich sind. Ein Teil davon sind die hier beschriebenen Standard-Funktionen, die für alle Objekte analoge Wirkung haben.

Für y ist jeweils ein Kennbuchstabe für den Objekttyp einzusetzen (p = Prozeß, q = Queue, d = Driver, t = Timer, m = Pipe (Mailbox), r = Semaphore, e = Event).

y_create (name, ...)

Mit dieser Funktion wird von einem Programm ein CMS-Objekt mit dem angegebenen Namen angelegt und ein Handle zurückgegeben. Alle CMS-Objekte können, sobald benötigt, dynamisch zur Laufzeit angelegt und auch wieder aus CMS entfernt werden (siehe y_delete()).

Das Anlegen von CMS-Objekten setzt ein geladenes und gestartetes Programm voraus, zum Beispiel wird ein CMS-Prozeß von dem Programm erzeugt, in dem sich der Programmcode dieses Prozesses befindet.

Neben dem Namen werden vom rufenden Programm über zusätzliche Funktionsparameter die gewünschten Attribute des zu generierenden Objektes festgelegt. Der Name ist in der Länge durch die Konstante NAMELEN beschränkt, deren Wert in CMS.H definiert ist.

y_open (name)

Mit der y_open()-Funktion erfolgt über den Namen der erste Zugriff auf ein existierendes, d.h. mit y_create() erzeugtes, CMS-Objekt.

Erst wenn dieser Aufruf erfolgreich ist und ein gültiges Handle für das Objekt geliefert hat, ist sichergestellt, daß das adressierte Objekt existiert und daß auf dieses zugegriffen werden kann. In allen weiteren Aufrufen wird das Objekt über sein Handle identifiziert. Bei jedem y_open() auf ein Objekt wird ein Zähler (Open-Zähler) inkrementiert, der die Anzahl der Nutzer des Objekts angibt. Jedem y_open() muß, wenn das Objekt nicht mehr benötigt wird, ein y_close() folgen. Ein noch geöffnetes Objekt kann nicht gelöscht werden.

y_delete (yhandle)

Mit der y_delete()-Funktion kann ein einmal angelegtes CMS-Objekt wieder aus CMS entfernt werden. Dabei wird überprüft, ob das Objekt von anderen Prozessen noch genutzt wird (Open-Zähler > 0). Das Objekt wird nur entfernt, wenn dieser Zähler = 0 ist, ansonsten wird das Objekt nur zum Löschen vorgemerkt.

y_close (yhandle)

Mit der Funktion y_close() wird der Zugriff auf ein CMS-Objekt beendet. Ein weiterer Zugriff ist erst nach erneutem y_open() möglich. Bei jedem y_close() auf ein Objekt wird sein Zähler dekrementiert, der die Anzahl der Nutzer des Objekts angibt. Falls der Zähler = 0 wird und ein y_delete()-Aufruf für dieses Objekt bereits erfolgt ist (erkennbar am gesetzten Delete-Flag), wird das Objekt gelöscht. CMS sichert, daß nur Objekte gelöscht werden, die von keiner Instanz mehr genutzt werden. Daher muß jedem y_open() ein y_close() folgen, wenn das Objekt nicht mehr benötigt wird.

y_getcb (yhandle, ycb)

Mit der Funktion y_getcb() können der aktuelle Zustand eines CMS-Objektes und seine Attribute abgefragt werden. Die Informationen werden in einem Info-Block zurückgegeben, der spezifisch für den Typ des Objektes ist.

y_handle (yhandle)

Mit der Funktion y_handle() wird zu einem Objekt-Handle das nächste aus der Liste aller Handles eines Objekttyps geliefert. Mit Hilfe dieser Funktion ist es möglich, eine Liste aller CMS-Objekte eines Typs zu erstellen (z.B. zu Diagnose- und Testzwecken).

y_list (void)

Mit der Funktion y_list() wird eine Liste der im System vorhandenen Objekte eines Typs ausgegeben. Die charakterisctischen Angaben zu den Objekten werden in einem Standard-Format ausgegeben.

Die folgenden beiden Funktionen sind für die Objekttypen Timer, Semaphore, Event, Queue und Pipe definiert:

y_status (yhandle)

Der Info-Block, der den vollständigen Zustand eines CMS-Objektes enthält, ist normalerweise groß. Oft wird nur ein spezieller Zustandswert benötigt, zum Beipiel der aktuelle Wert eines Timers. Die Funktion y_status() liefert für das angegebene Objekt einen für den Objekttyp charakteristischen Zustandswert.

y_signal (yhandle, kind, signal)

Mit dieser Funktion können bestimmte Operationen auf CMS-Objekten mit einem Signal verknüpft werden. Eine bestimmte Operation auf dem mittels "yhandle" identifizierten CMS-Objekt führt zum Senden des Signales "signal" an den Prozeß, der y_signal() aufgerufen hat.

Die Operation, die zum Senden des Signales führt, ist für den Objekttyp spezifisch. Es ist in jedem Fall eine Operation, auf die ein Prozeß unter Umständen warten muß (z.B. Erscheinen von Daten in einer Queue oder Ablauf eines Timers). Durch Kombination solcher Signale im Signalwort des Prozesses läßt sich in einfacher Weise ein Multiple Wait realisieren.

2. CMS-Objekte

2.1. Prozesse

Prozesse sind Programme, die beim Betriebssystem als Prozesse angemeldet sind und am Multitasking teilnehmen und denen entsprechend ihrer Priorität Prozessorkapazität zugewiesen wird (Scheduling). Jeder Prozeß hat seinen eigenen Datenbereich und seinen eigenen Stack.

Prozesse konkurrieren um Systemressourcen und sind in der Lage, quasi parallel und asynchron verschiedene Vorgänge abzuarbeiten. Die Priorität von CMS-Prozessen ist dynamisch veränderbar, um in bestimmten Betriebszuständen z.B. zeitkritische Vorgänge vorrangig bearbeiten zu können.

Prozesse kommunizieren miteinander über die von CMS angebotenen Mechanismen zur Interprozeß-Kommunikation (IPC). Falls Prozesse aus ein bestimmtes Ereignis warten, werden sie vom System solange syspendiert, bis das Ereignis eingetreten ist oder bis eine vom Prozeß gewählte Wartezeit abgelaufen ist.

Prozesse sind im laufenden System dynamisch nachladbar, d.h. sie werden nicht mit dem Betriebssystem zusammengelinkt, sondern in Form von einzelnen Komponenten. Sie können so ins ROM des Embedded Systems gebrannt oder von dort ins RAM geladen werden oder über eine Testschnittstelle zur Laufzeit nachgeladen werden. In jedem Fall werden sie einzeln gestartet.

2.1.1. Signale für CMS-Prozesse

Signale sind Mechanismen zur Benachrichtigung und Synchronisation der Prozesse in einem Multitasking-System. Signale teilen allein durch die Tatsache, daß sie gesetzt sind, das Eintreten eines Ereignisses mit. Darüber hinaus enthalten sie keine Information, insbesondere nicht die Identität des Senders.

Jedem Prozeß wird von CMS ein Signalwort (32 bit) zugeordnet, wobei jedes Signal einem Bit dieses Wortes entspricht. Vier Signale und ihre Position im Signalwort werden von CMS vordefiniert. Die restlichen Signale können frei verwendet werden. Das Signalwort kann in einer unteilbaren Operation abgefragt und neu gesetzt werden (Funktion p_getset()).

Prozesse können sich in den Wartezustand versetzen und Ereignisse definieren, auf die sie warten (Daten liegen in einer Queue vor, Timer ist abgelaufen, Daten wurden vom Treiber eingelesen), und bei deren Eintreten sie aktiviert werden. Das geschieht, indem sie die Signalbits bezeichnen, auf die sie reagieren wollen. Auf diese Weise kann auch ein Multiple Wait realisiert werden. Das wird immer dann benötigt, wenn ein Prozeß gleichzeitig auf das Eintreten verschiedener Ereignisse wartet (z.B. Ablauf eines Timers oder Eintreffen einer Nachricht in einer Queue) und beim Eintreffen eines dieser Ereignisse benachrichtigt werden soll (logisches Oder).

2.2. Timer

Zur Bearbeitung zeitlich bedingter Aufgaben können in CMS Timer definiert werden, z.B. zum Auslösen von Aktionen nach Ablauf einer vorgegebenen Zeitspanne. Timer sind Zähler, die mit einem Initialwert gestartet und mit jedem Systemtick (systemspezifische Zeiteinheit) dekrementiert werden. Die Auflösung der Ticks in Millisekunden ist hardware-abhängig.

Nach dem Ablaufen eines Timers (Erreichen der Null) wird der Prozeß, der den Timer gestartet hat, mit einem Signal benachrichtigt oder es wird eine vom Nutzer definierte (Timeout-)Funktion aufgerufen.

Wahlweise bleibt der Timer dann (One-Shot-Timer) oder wird automatisch wieder gestartet (periodischer Timer).

2.3. Semaphoren

Semaphoren sind Mechanismen zum gegenseitigen Ausschluß (mutual exclusion) von Prozessen beim konkurrierenden Zugiff auf Ressourcen. Ein Prozeß, der eine Ressource exklusiv benutzen will, fordert die Semaphore an (r_request()) und sperrt damit für alle anderen den Zugriff, bis die Semaphore wieder freigegeben wird (r_release()). Dies kann durch das Setzen eines Signals angezeigt werden.

CMS stellt sicher, daß das Anfordern der Semaphore nur für einen Prozeß zur Zeit erfolgreich ist; die anderen Prozesse warten bei r_request(), bis der vorhergehende Prozeß r_release() ausgeführt hat. Wartende Prozesse erhalten die Semaphore in der Reihenfolge, in der sie sich angemeldet haben.

Die Prozesse müssen sich kooperativ verhalten und die Ressource nur dann benutzen, wenn sie die Semaphore besitzen, die sie schützt.

Innerhalb eines Prozesses kann eine Semaphore mehrmals angefordert werden, ohne daß sie zwischendurch freigegeben werden muß. Dazu wird ein Zähler mitgeführt und der gegenwärtige Inhaber der Semaphore gemerkt. Die Semaphore wird frei, wenn dieser Zähler 0 ist.

2.4. Events

Events sind von CMS verwaltete globale Variablen (jeweils 16 bit), die von allen Prozessen gesetzt und abgefragt werden können. Außerdem können Prozesse auch auf das Eintreten bestimmter Event-Werte warten. Damit sind Synchronisation und in begrenztem Maße auch Kommunikation zwischen Prozessen möglich.

Beim Setzen und Warten können verschiedene arithmetische und Bit-Operatoren auf den Event-Wert angewandt werden. Damit sind zum Beispiel auch Counter oder Mischformen aus Counter und Signal realisierbar.

Events können in einer unteilbaren Operation abgefragt und neu gesetzt werden (e_getset()).

CMS kann veranlaßt werden, beim Setzen eines Events einen Prozeß davon zu benachrichtigen. Dazu definiert der Prozeß mit e_signal() einen Signalwert und wartet mit p_wait(), bis dieser Wert in seinem Signalwort gesetzt wird.

2.5. Queues

Queues im Sinne von CMS sind FIFO-Speicher, die der Kommunikation zwischen Prozessen mit Messages dienen. Messages sind beliebige Bytefolgen. Sie können nur blockweise geschrieben bzw. gelesen werden (entweder vollständig oder gar nicht).

Die Übertragung der Daten kann wahlweise durch das Kopieren aller Bytes einer Message (Kopier-Modus) oder durch die Übergabe der Anfangsadresse (Pointer-Modus) erfolgen. Im Pointer-Modus sollten die Prozesse aber zusätzlich über Semaphoren komunizieren, um sicherzustellen, daß ein Datenbereich nicht vom Versender überschrieben wird, bevor der Empfanger ihn gelesen hat. Wenn die Queue im Kopier-Modus betrieben werden soll, müssen beim Anlegen der Queue Anzahl und maximale Größe der Messages angegeben werden, damit CMS entsprechend Speicher reserviert. Im Pointer-Modus wird nur die Anzahl der Messages benötigt.

Die Messages können sowohl mit normaler Priorität (= 0) übermittelt werden als auch mit erhöhter Priorität. Dann werden sie innerhalb der Queue vor alle niedriger priorisierten Messages, aber hinter alle höher oder gleich priorisierten gestellt.

In zwei Fällen wird der rufende Prozeß blockiert: Beim Versuch, aus einer leeren Queue zu lesen, und beim Versuch, eine Message zu schreiben, die die Queue nicht mehr aufnehmen kann. Durch die Angabe eines Timeout-Wertes beim Aufruf der Read- oder Write-Funktion wird die Wartezeit begrenzt.

2.6. Pipes

Pipes sind - wie die Queues- FIFO-Speicher. Der Unterschied besteht darin, daß hier in “beliebiger” Länge geschrieben bzw. gelesen werden kann, d.h. die Längen beim Lesen und Schreiben dürfen unterschiedlich sein. Über die Strukturierung der Information müssen sich Sende- und Empfangsprozeß natürlich irgendwie einig sein.

Die Länge des Puffers sollte eine Zweierpotenz sein, sonst wird sie automatisch auf die nächstgrößere Zweierpotenz erhöht.

2.7. Driver

Driver sind Programme, die nicht wie Prozesse am Scheduling teilnehmen, sondern Funktionen exportieren, die von Prozessen (oder auch Drivern) wie Unterprogramme aufgerufen werden, also eher gemeinsam nutzbare Bibliotheken darstellen. CMS stellt den Mechanismus zur Verfügung, um die Export-Funktionen dieser eigenständigen Programme so aufzurufen, als wären sie an das rufende Programm angelinkt.

In CMS werden Communication-Driver und einfache Driver (Server) unterschieden.

Die Communication-Driver dienen dem Datentransport, beispielsweise zum Zugriff auf die Hardware unabhängig von der jeweiligen technischen Ausprägung. Sie realisieren eine Schnittstelle der Inter-Prozeß-Kommunikation (IPC), das sind die Funktionen xyz_read() / xyz_write() / xyz_look() / xyz_flush() / xyz_signal()), wobei für xyz für den Namen des Treibers steht.

Um mit einem Driver mehrere gleichartige Schnittstellen erfassen zu können, haben die Driver-Funktionen den Parameter 'device', der die Nummer der Schnittstelle angibt.

2.8. IPC / Redirection

Queue, Pipe und Communication Driver sind Objekte, die der Inter-Prozeß-Kommunikation dienen. Sie alle realisieren eine Standard-Schnittstelle zum Datentransport - die Funktionen y_read(),y_write(),y_look(),y_flush() und y_signal() (mit y = q für Queues, m für Pipes und d für Driver).

Mit diesem Standard-Interface ist das Umlenken von Schreiboperationen verschiedener Objekttypen der IPC möglich, z.B. können Daten, die mit q_write() geschrieben werden, an eine physikalische Schnittstelle umgelenkt werden, für die normalerweise ein Aufruf von d_write() benutzt wird.

Redirection bedeutet, die zu übertragenden Daten selektiv an andere Ziele umzulenken. Alle oder ausgewählte Messages, die in eine Queue geschrieben werden, werden bei der Redirection zusätzlich oder ausschließlich an ein anderes Ziel weitergeleitet oder auch verworfen. Das Zielobjekt kann eine oder mehrere Queues, Pipes oder Communication Driver sein. Dasselbe kann beim Schreiben in eine Pipe oder einen Communication Driver vorgenommen werden.

Die notwendige Verdoppelung bzw. Umlenkung der Daten wird vollständig von CMS vorgenommen. Die Einstellung der Redirection erfolgt durch die Funktion y_control().

Die Funktion y_control() erfordert eine bestimmte Datenstruktur, in der die Umlenkungsparameter beschrieben sind.

3. Programme unter CMS

3.1. Prozesse

3.1.1. Programmstruktur von CMS-Prozessen

Programme, die unter CMS laufen sollen, sind prinzipiell wie übliche C-Programme aufgebaut, allerdings sind Vorkehrungen zu treffen, damit mehrere Programme gestartet und quasigleichzeitig abgearbeitet werden können. Um als Prozesse zu laufen, muß ein Programm

1. sich bei CMS als Prozeß anmelden.
   ==> CMS kann dem Prozeß regelmäßig CPU-Zeit zuweisen.
2. im Speicher verbleiben.
   ==> Bei jeder Zuweisung von CPU-Zeit an diesen Prozess steht immer noch dasselbe Programm im Speicher.

Damit ein Programm im Speicher bleibt, gibt es zwei Möglichkeiten:

1. Es enthält eine Endlosschleife, in der es periodisch Aufgaben ausführt, ohne sich zu beenden.
2. Es macht sich resident. Dafür gibt es CMS ein Makro namens RETURN, das diese Aufgabe in plattformspezifischer Weise erledigt.

Zum Aufruf der CMS-Funktionen aus den Anwendungsprogrammen heraus wird ein spezieller Mechanismus benutzt:

Jede CMS-Funktion hat eine Funktionsnummer, die in CMS.H festgelegt ist. Ferner gibt es in CMS.H Makros, die die Aufrufe der CMS-Funktionen abbilden auf den Aufruf der zentralen Einsprungfunktion von CMS. Dabei wird die Funktionsnummer als zusätzlicher erster Parameter eingefügt. Nach dieser Nummer wird innerhalb von CMS wieder verzweigt.
Das Anwendungsprogramm muß nur diese eine Funktion kennen und aufrufen können. Diese Funktionalität ist für alle Anwendungsprogramme auf demselben System gleich und liegt im Modul CMA fertig vor. Daher müssen Programme für CMS mit cma.obj zusammengelinkt werden.

3.1.2. Beispiel

Das Programms p2.c zeigt die Struktur eines einfachen CMS-Programms mit mehreren Prozessen; es wird im Anschluß erläutert.

```
#include <stdio.h>
#include "cms.h"
void proc1();
void proc2();
int th;


main()
{
int p1, p2, p3;
long y;
unsigned long t;
```

```
CMS_HANDLE h;

   long y = 0;
   p1 = p_create( "proc1", proc1, 1, 1, 4000 );
   p2 = p_create( "proc2", proc2, 1, 2, 4000 );
   p3 = p_create( "proc3", proc2, 1, 3, 4000 );
   printf("p1 = %d , p2 = %d, p3 = %d\n", p1, p2, p3 );
   p_start( p1 );
   p_start( p2 );
   p_start( p3 );

   for(;;)
   {
      y++;
      if( y == 20000000 )
      {
         y = 0;
         s_mode( TIMESLICE );
         s_list();
         printf("process handles: \n");
         h = 0;
         do
         {
            printf(" %X -> ", h );
            h = p_handle( h );
            printf("%X\n", h );
         } while( h != 0 );

         printf("(end of process handles)\n");
         printf("timer handles: \n");
         h = 0;
         do
         {
            printf(" %X -> ", h );
            h = t_handle( h );
            printf("%X\n", h );
         } while( h != 0 );

         printf("(end of timer handles)\n");
         p_priority( 1, 0 );
      }
   }
}



void proc1( long x )   /* process PROC1 */
{
int y, rc;
CMS_TIME t;
CMS_TIME s;

   y = 10000 * x;
   th = t_create( "proc1time" );
   t_start( th, 1000, tout, 1, 0 );
   for(;;)
   {
      y++;
      rc = p_sleep( 7300 );
      s_time( &t );
      t_status( th, &s );
      printf("PROC1 rc=%d th=%X (status: %ld) x=%ld y=%d time: %ld\n",
              rc, th, s, x, y, t );
   }
}
```

```
void tout( p )        /* timeout function */
long p;
{
   printf(" //%ld// ", p );
   t_start( th, 1000, tout, p+1, 0 );
}


void proc2( long x )    /* process PROC2 */
{
int y, rc;
CMS_TIME t;

   y = 10000 * x;
   for(;;)
   {
      y++;
      rc = p_sleep( 13000 );
      s_time( &t );
      printf("PROC2 rc=%d x=%ld y=%d time: %ld\n", rc, x, y, t );
      if( x == 3 )
      {
         s_list();
         p_list();
         t_list();
         s_mode( TIMESLICE );
      }
   }
}
```

Das vorliegende Beispiel enthält ein einziges Programm, das interne Funktionen als weitere Prozesse startet und selbst im Speicher verbleibt, ohne sich resident zu machen, also mit Endlos-Schleife.

Die Include-Datei cms.h muß in jedem Fall eingebunden werden. Sie bewerkstelligt die Abbildung der CMS-Funktionen auf die Einsprungfunktion von CMS.

Die main()-Funktion kreiert drei Prozesse nach dem Muster

```
p2 = p_create( "proc2", proc2, 1, 2, 4000 );
```

Damit hat erhält CMS die Adresse der Einsprungfunktion eines Prozesses und macht einen Eintrag in der Prozeßliste. Noch steht der Prozeß. Um ihn tatsächlich zu aktivieren, ist das explizite Starten notwendig:

```
p_start( p2 );
```

Anschließend geht die main()-Funktion zu einer Endlosschleife über, in der sie einen Zähler hochsetzt. Wegen des time-slicing erhält main() immer wieder die CPU; ein expliziter Aufruf von p_schedule() ist hier nicht nötig. Nach 20000000 Prozeßwechseln gibt main() eine Liste der Prozeß und Timer-Handles aus und beginnt von neuem zu zählen.

Die Funktionen proc1() und proc2() werden als Prozesse bei CMS angemeldet und gestartet. Wesentlich ist, daß sie jeweils eine Endlosschleife enthalten. Darin setzten sie einen Zähler hoch und legen sich dann schlafen mit

```
rc = p_sleep( 13000 );
```

CMS sorgt dafür, daß sie nach der angegebenen Wartezeit wieder am Scheduling teilnehmen. Während alle anderen Prozesse schlafen, bekommt die Schleife in der main()-Funktion CPU-Zeit (Idle-Prozeß, s.u.). In

einem weniger trivialen Beispiel kann die Hauptschleife eines Anwendungsprozesses z.B. in einem q_read() warten.

Wenn proc1() aufwacht, stellt er mit t_status() den Timer-Status fest (seine Restlaufzeit) und gibt ihn in der Meldung "PROC1 rc= .." aus. Dann durchläuft er die Schleife erneut.

proc2() wird zweimal gestartet mit unterschiedlichen Parametern. Jeder der beiden entstehenden Prozesse gibt in seiner Hauptschleife in der Meldung "PROC2 rc= .." die Systemzeit aus. Die zweite Inkarnation von proc2() gibt zusätzlich Prozeß- Timer- und System-Listen mittels der entsprechenden CMS-Funktionen aus.

proc1() startet zusätzlich einen Timer mit der timeout-Funktion `tout()`:

```
t_start( th, 1000, tout, 1, 0 );
```

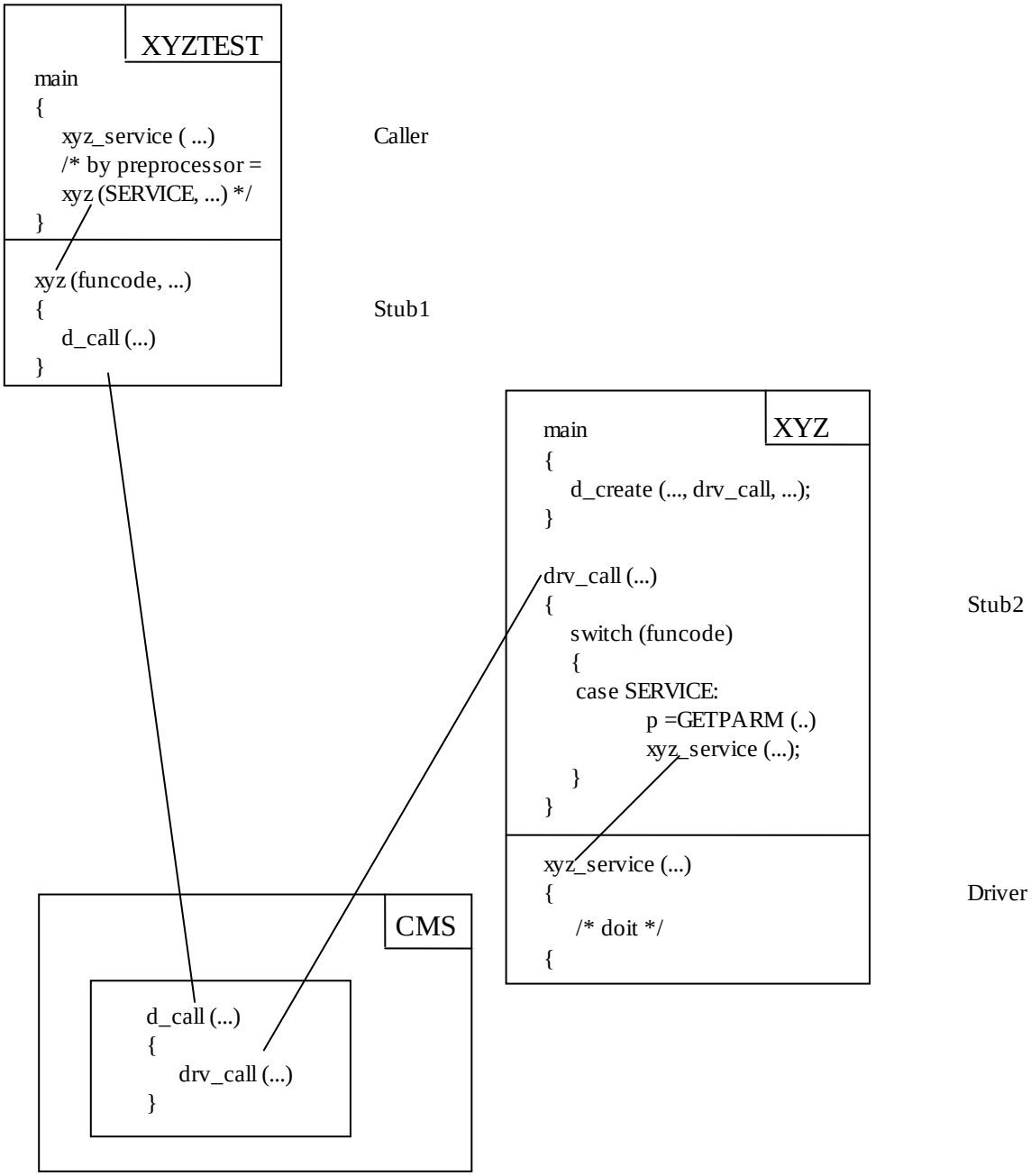
Die Timeout-Funktion wird von CMS unabhängig von den drei Prozessen aufgerufen, sobald der Timer abgelaufen ist. Sie gibt einen Zähler aus, in // // eingeschlossen und startet den Timer erneut. Das Beispiel zeigt, daß innerhalb der Timeout-Funktion Funktionen ohne Einschränkungen aufgerufen werden können.

Die Ausgaben der Listen, Zähler und Meldungen können sich gegenseitig unterbrechen.

3.2. Driver

3.2.1. Aufrufprinzip

Der Aufruf von Drivern geschieht ähnlich dem Remote Procedure Call (RPC). Beim RPC gibt es einen Stub (ein angelinktes Programmstück) unterhalb des Aufrufers, der den Driver simuliert, und einen Stub oberhalb des Drivers, der den Aufrufer simuliert. Die beiden Stubs erledigen die notwendige Kommunikation, hier mittels CMS (Erläuterung umseitig.)



Alle Driverfunktionen werden im Anwendungsprogramm mit ihrem richtigen Namen aufgerufen. Durch den Präprozessor werden sie mittels Makros auf den Aufruf einer einzigen Funktion abgebildet, die als zusätzlichen Parameter die Funktionsnummer (function code) hat. Im Bild ist das

xyz_service(...) --> xyz (SERVICE, ...).

Diese zentrale Funktion xyz() ruft die CMS-Funktion d_call() in CMS auf und übergibt ihr die Parameter, genauer den Funstionscode und die Adresse des Parameterstacks. xyz() bildet den Stub auf der Aufruferseite.

CMS kennt den Eintrittspunkt drv_call() des Drivers vom d_create() her und springt dorthin. Dabei werden die Parameter von d_call() an drv_call() übergeben. Die Funktion drv_call() ihrerseits verzweigt nach dem Funktionscode und ruft die Driverfunktion mit ihrem richtigen Namen und den wieder aus dem Stack herausgelösten Parametern auf. Sie bildet also - zusammen mit der main()-Funktion - den Stub auf der Server-Seite.

Dabei sind Datensegement-Umschaltungen notwendig. Sie werden von CMS erledigt. Weder der Driver noch das Anwendungsprogramm merken etwas davon. Für den Aufrufer sieht es so aus, als wären die Driverfunktionen angelinkt.

3.2.2. Programmstruktur eines Drivers

Um einen Driver unter CMS zu schreiben, sind keine besonderen programmtechnischen Tricks nötig. Viele Bestandteile liegen als Bausteine fertig vor. Dies sind cms.h, drvmain.c, drvmain.h, drvapi.c, drvapi.h.

Der Driver-Rumpf ist nichts anderes als ein normaler Programm-Modul (ohne main()). Durch den CMS-Aufrufmechanismus kann er sogar seine Export-Funktionen als static deklarieren.

Neben den Funktionen, die den spezifischen Zweck des Drivers ausmachen, muß der Driver einige Funktionen implementieren, deren Vorhandensein CMS voraussetzt und die zum Teil automatisch aufgerufen werden. Sie werden zusammen als Standard-Driver-Funktionen bezeichnet. Wenn XYZ ein beliebiger Driver ist, sind das die Funktionen xyz_init(), xyz_exit(), in einigen Fällen, die später noch erläutert werden, kommen xyz_open() und xyz_close() hinzu, und Communication Driver müssen außerdem xyz_read(), xyz_write(), xyz_look(), xyz_flush(), xyz_signal() implementieren.

Das folgende Beispiel verwendet D2 als konkreten Driver anstelle des allgemeinen XYZ. Der Modul d2.c zeigt den Aufbau eines Drivers:

```
/* sample driver D2 */

#define DRV    d2                              /* driver properties: */
#define FLAGS  CMSDRV_NONSHARED | CMSDRV_MONITOR
#define DEVS    0
#define MAXFUN 20

#include "cms.h"
                                           /* extra function codes: */
#include "d2.h"
                                      /* extra function prototypes: */
CMS_RETURN d2_store( int lng, char pattern );
CMS_RETURN d2_restore( int * lng, char * buf );

static int  length;
#define D2BUFLEN   200
static char d2buf[D2BUFLEN];

                                 /* main, with drv_call() to complete: */
#include "drvmain.c"
                                           /* extra function cases: */
   case D2_STORE:
   {
   int lng;
   char pattern;

      lng = GETPARM(int,parm);
      pattern = GETPARM(char,parm);
      rc = d2_store( lng, pattern );     /* rc  defined in drvmain.c  */
      break;
   }
   case D2_RESTORE:
   {
   int *lng;
   char *buffer;

      /*GETPARM(int,parm); */              /* discard 'dev' parm on stack*/
      lng = GETPARM(int *,parm);
      buffer = GETPARM(char *,parm);
      rc = d2_restore( lng, buffer );
      break;
   }

#include "drvcend.c"                    /* end of drv_call()         */

/* standard funtions: */
CMS_RETURN d2_init( void )
{
   memset (d2buf, 0, sizeof (d2buf) );
   return CMS_OK;
}

.... /* hier stehen d2_open(), d2_close(), d2_exit(), d2_monitor(),... */

                                                 /* extra functions: */
CMS_RETURN d2_store( int lng, char pattern )
{
if (lng > D2BUFLEN)
       return (CMS_ERROR);
   memset (d2buf, pattern, lng);
   d2buf[lng] = 0;
   length = strlen (d2buf);
   return CMS_OK;
}     /* end d2_store() */


CMS_RETURN d2_restore( int * lng, char * buf )
{
```

```
char *b;
int i;

   if (strlen (d2buf) != length)
       return (CMS_ERROR);
   b = d2buf;
   for (i=0; i<= length; i++)
      *buf++ =*b++;
   *buf = 0;
   *lng = length;
   memset (d2buf, 0, D2BUFLEN);
   return CMS_OK;
}     /* end d2_store() */
```

Im folgenden werden die Statements näher erläutert.

#define DRV drivername

damit wird der Präprozessor veranlaßt, die Driver-Bausteine an den speziellen Driver anzupassen. Die Zeichenfolge “DRV/**/” in den weiteren Deklarationen wird z.B. durch den Drivernamen “d2” ersetzt.

#define FLAGS .....

mit diesen Flags wird der Driver konfiguriert, s.u.

#include "d2.h"

hier werden die Funktionscodes der Nicht-Standard-Funktionen des Drives definiert

#include drvmain.c

Der Module drvmain.c enthält

1. die Prototypen der Standard-Driver-Funktionen, die dort als DRV/**/_write(..) etc. erscheinen. Sie werden mit Hilfe des “#define DRV d2” an den echten Drivernamen “d2” angepaßt

2. die main()-Funktion des Treibers .
   Sie ruft d_create() auf zur Registration des Treibers bei CMS. Dabei teilt sie CMS den Namen “d2” und die Einsprungadresse drv_call() mit.
   Im Makro RETURN macht sich der Driver resident. Es ist in CMS_TYPES.H CPU-spezifisch definiert.

3. den Anfang der Funktion drv_call().
   Sie stellt einen großen Switch über den Funktions-Code dar.
   Einzelne Funktionen werden durch die Flags ein- oder ausgeblendet, z.B. die monitor()-Funktion.

Bis hierher ist alles vorgefertigt. Nun muß die Verzweigung zu den übrigen, nichtstandardisierten Funktionen des Drivers folgen, jede in einem “case”-Block. Für diese Funktionen müssen in einer driver-spezifischen Include-Datei “d2.h” die zusätzlichen Function Codes definiert sein, falls der Driver mehr als das Standard-Interface anbietet, im Beispiel `D2_STORE` und `D2_RESTORE`.

Da die Funktionsparameter per Stack-Adresse übergeben werden, müssen sie, um sie einzeln an die Driver-Funktionen weiterreichen zu können, von Stack abgesammelt werden. Wie das zu geschehen hat, ist hardware-abhängig, deshalb wird dafür das Makro GETPARM verwendet. Der Aufruf

par = GETPARM (typ, sp)

bewirkt folgendes:

GETPARM liest einen Parameter vom angegebenen Typ vom Stack und liefert ihn als Resultat zurück. Als Typ können alle Datentypen angeben werden. Der Stackpointer wird durch das Makro weitergestellt.
Beispiel:

```
dh = GETPARM (CMS_HANDLE, sp);
field = GETPARM (char **, sp);
```

Nach allen Verzweigungen wird der Switch mit
#include drvend.c
abgeschlossen und damit auch der Server-Stub.

Dann folgt die Implementierung alle Driver-Funktionen, sowohl der Standard-Driver-Funktionen als auch der Extra-Funktionen.

Der Driver wird zusammen mit cma.obj gelinkt. Der Modul CMA enthält den Aufruf der Einsprungfunktion von CMS. Hier ein Ausschnitt aus dem Makefile:

```
d2.exe: d2.obj cma.obj
        $(LINK)

d2.obj: d2.i
        $(DCC)

d2.i: ..\d2.c ..\d2.h ..\cms.h cmstypes.h ..\drvmain.c ..\drvcend.c
        $(CPP)
        $(CPP2)

d2api.obj: ..\d2api.c ..\cms.h cmstypes.h ..\drvapi.c
        $(CC1)
```

Hierbei bedeuten:

`$(LINK)`	Aufruf des Linkers
`$(DCC)`	Aufruf des Compilers, Quelle ist *.i
`$(CPP)`	Aufruf des Präprozessors von *.c nach *.i, Substitution
`$(CPP2)`	Aufruf des Präprozessors von *.i nach *.ii, Substitution
`$(CC1)`	Aufruf des Compilers, Quelle ist *.c

mit den entsprechenden Optionen.

3.2.3. Verwendung der Driver-Funktionen

Ein Programm, das den Driver benutzen möchte, kann davon ausgehen, daß er bereits aktiv ist. Wenn der Driver nicht mehrere Devices verwaltet, kann ohne weitere Vorbereitung eine Driver-Funktion aufgerufen werden. Die notwendige Verbindung zum Driver stellt die zentrale Funktion xyz() im Stub drvapi.c automatisch beim ersten Aufruf her, d.h. d_open() wird implizit aufgerufen.
Die Funktion d_open()muß nur dann explizit aufgerufen werden, wenn der Driver mehrere Devices verwaltet. Dann muß natürlich ein d_close() am Ende stehen.

Das Programm D2TEST vedeutlicht den Aufruf von Driver-Funktionen.

```
/* sample test program for driver d2 */

#include "d2api.h"
#define TIME 5000
#define PLAYERS  4   /* number of processes joining the game */
#define STORELEN 16

static char pattern [PLAYERS] =    { 'A', 'B', 'C', 'D'};
void user( long x);

main()
{
int dh, p, s;
char rbuf[100];
CMS_HANDLE proc[5] = {-1, -1, -1, -1, -1};
char *name[5] = { "user1", "user2", "user3", "user4", "user5"};
CMS_HANDLE p1, p2=-1, p3=-1;
long y;

   printf("This is D2TEST.\n");
   y = 0;
   for (p=0; p< PLAYERS; p++)
   {
     proc[p] = p_create( name[p], user, 1, p+1, 4000 );
       if (proc[p] > 0) p_start (proc[p]);
   }

   for(;;)
   {
      if (++y == 1000000)
      {
         d_list();
         y = 0;
       }
      p_schedule();
   }
}  /* end main() */


void user( long proc)
{
int i, j, h, lng, p, rc, turn;
char rbuf[100], *b;

   printf("User%ld started.\n", proc);
   h = d_open ("d2", 0);
   for (turn = 0; turn < 100; turn++)
   {
      rc = d2_store ( STORELEN, pattern[proc-1] );
      if (rc < 0)
         printf("User%ld; store (%c) => %d\n", proc, pattern[proc-1], rc );
      p_schedule();
```

```
      rc = d2_restore ( &lng, rbuf );
      if (rc < 0)
         printf("User%ld restore (%c)  => %d\n", proc,pattern[proc-1], rc);
      else
      {
         if (lng != STORELEN)
            printf("User%ld restore : ERROR;  lng=%d\n", proc, lng );
         else
         {
            b = rbuf;
            p = pattern[proc-1];
            for (i=0; i<lng; i++)
            {
               if (*b != p)
               {
                  printf(
                  "User%ld restore ERROR: bad byte '%c' instead of '%c'\n",
                  proc, *b, p );
                  break;
               }
               b++;
            }
            printf("<%ld>", proc );
         }
      }
      p_schedule();
   }

   for (j=0; j<10; j++)
      p_schedule();
   d_close (h);
   p_delete ( p_pid() );
}  /* end user() */
```

#define DRV “d2”

dient wie oben zur Substitution von “DRV/**/” durch den spezifischen Drivernamen.

#include “d2.h”

Diese Driver-spezifische Header-Datei definiert zusätzliche Funktionsnummern, z.B. `D2_STORE` und `D2_RESTORE`.

#include “d2api.h”

Die Driver-spezifische Include-Datei enthält Abbildungen der nicht-standardisierten Driver-Funktionen auf eine einzige zentrale Funktion d2() mittels Funktionsnummern (Function Codes). Z.B. kommt in d2api.h. vor:

#define d2_store(a,b) d2(D2_STORE, a, b)

drvapi.c enthält den Stub und wird angelinkt.

Die zentrale Funktion liegt als Muster mit dem Funktionsnamen DRV/**/() in drvapi.c vor. Durch den Präprozessor wird der Name durch den aktuellen Driver-spezifischen Funktionsnamen d2() ersetzt.

Der folgende Ausschnitt aus dem Makefile zeigt, wie ein Programm gelinkt wird, das Driverfunktionen benutzt. Es gelten dieselben Makros wie oben.

```
d2test.exe: d2test.obj d2api.obj cma.obj
```

```
        $(LINK)

d2api.obj: ..\d2api.c ..\cms.h cmstypes.h ..\drvapi.c
       $(CC1)

d2test.obj: d2test.i
        $(DCC)

d2test.i: ..\d2test.c ..\d2api.h ..\d2.h ..\cms.h cmstypes.h ..\drvapi.h
        $(CPP)
        $(CPP2)
```

3.2.4. Driver-Konfigurationen

Beim Kreieren eines Drivers lassen sich über Flags verschiedene Driver-Eigenschaften einstellen. Im obigen Beispiel sind sie in der Konstanten FLAGS festgelegt und werden durch den Aufruf von d_create() innerhalb des Moduls drvmain.c an CMS übergeben. Jede Eigenschaft wird durch ein Bit in FLAGS repräsentiert; die Bits können kombiniert werden. Folgende Eigenschaften sind möglich:

Datenmodell:	Shared vs. Non_Shared (Flag CMSDRV_NONSHARED gesetzt)
Verwendungszweck.	Communication Driver (Flag CMSDRV_ IPC gesetzt) vs. einfacher Driver
Schnittstellen:	mehrere Devices (Flag CMSDRV_ DEVICES gesetzt) vs. kein oder ein Device
Monitoring:	die Funktion xyz_monitor() wird eingeblendet (Flag CMSDRV_MONITOR gesetzt) oder ausgeblendet

Daraus ergeben sich Konsequenzen sowohl für den Funktionsbestand als auch für die Inhalte der Funktionen. Die Einstellungen können in beliebigen Kombinationen miteinander vorkommen. Bei den folgenden Erläuterungen ist ´xyz´durch den Namen des jeweiligen Drivers zu ersetzen.

Shared Model (Flag CMSDRV_NONSHARED nicht gesetzt)

Im Shared Data Model hat der Driver ein einziges Datensegment, das für alle Aufrufe verwendet wird, unabhängig davon, welches Programm einen Aufruf ausgelöst hat, d.h. alle Aufrufer des Drivers teilen sich ein Driver-Datensegment. Der Driver selbst muß unterscheiden, welches Programm gerade bedient wird. Dazu erhält er beim Aufruf jeder Driver-Funktion die Prozeß-Nummer. Diese kann er als Index in seinen Datenfeldern benutzen.

Das Shared Data Model bietet sich für Communication Driver an, da dort oftmals statische Daten, Schnittstellenparameter etc. aufbewahrt werden müssen, die für alle Aufrufer gleich sind.

CMS sequentialisiert die Aufrufe mittels Semaphoren, so daß der Driver während der Abarbeitung einer Funktion nicht durch einen weiteren Funktionsaufruf unterbrochen wird.

Die init()- Funktion (z.B. xyz_init()) wird beim Kreieren des Drivers aufgerufen zur globalen (einmaligen) Initialisierung des Driver-Datensegments. Hier kann der Driver den für alle Nutzer gemeinsamen Datenbereich initialisieren.

Non Shared (Flag CMSDRV_NONSHARED gesetzt)

Das Non-Shared Data Model wird verwendet, wenn eine Funktionsbibliothek eingerichtet werden soll. In diesem Datenmodell erhält jedes Programm, das die Driver-Funktionen nutzt, ein eigenes Driver-Datensegment zugewiesen. Beim Aufruf einer Driver-Funktion nimmt CMS eine Segment-Umschaltung vor, so daß das gesamte Datensegment für die Belange der aktuellen Applikation zu Verfügung steht. Damit ist es natürlich auch nicht möglich, gemeinsame Einstellungen für verschiedene Aufrufer oder Devices aufzubewahren.

Die init()-Funktion wird von d_create() aufgerufen, in einem Stadium, in dem nur das Ur-Datensegment des Drivers vorhanden ist. Von diesem Datensegment wird bei jedem d_open() eine Kopie angelegt. Durch die Initialierung des Ur-Datensegments kann für die Einzel-Initialisierung unter Umständen Arbeit gespart werden.

Die Funktionen xyz_open() und xyz_close() (für den Driver XYZ) müssen bei diesem Datenmodell unbedingt implementiert sein. Die Funktion d_open() wird beim ersten Aufruf einer der Driver-Funktionen automatisch aufgerufen. Sie allokiert ein neues, aufruferspezifisches Datensegment, kopiert das Ur-Datensegment und ruft xyz_open() zur spezifischen Initialisierung. Analog wird xyz_close() zum Freigeben aufgerufen.

IPC (Flag CMSDRV_IPC gesetzt)

Mit diesem Flag wird vereinbart, daß der Driver ein Communication Driver ist, d.h er implementiert die Standard-IPC-Funktionen xyz_read(), xyz_write(), xyz_flush(), xyz_look() und xyz_signal(). Die Verzweigung zu diesen Funktionen wird im Server-Stub eingeblendet. Ist das Flag nicht gesetzt, kennt der Driver diese Funktionen nicht und kehrt beim Aufruf dieser Funktionen mit dem Fehlercode CMS_FUNCTION zurück. Für einen Communication Driver istRedirection möglich.

Devices (Flag CMSDRV_DEVICES gesetzt)

Das Flag besagt, daß der Driver mehrere Devices verwaltet, z.B. mehrere physische Schnittstellen. In diesem Fall wird der Parameter ***dev*** der Driver-Funktionen ausgewertet. Für jedes Device bekommt der Driver ein eigenes Handle. Daher muß eine Applikation die Driver-Management-Funktion d_open() explizit aufrufen. Der Driver muß die Standard-Funktionen xyz_open() und xyz_close() implementieren. Durch das Flag wird im switch von drv_call() die Verweigung zu diesen Funktionen eingeblendet. d_open() und d_close() aktivieren die entsprechenden Funktionen des Drivers.

Bei der Konfiguration “No Devices” (Flag CMSDRV_DEVICES nicht gesetzt) sind keine Besonderheiten zu beachten.

Monitoring (Flag CMSDRV_MONITOR gesetzt)

Mit diesem Flag wird die Verzweigung zur Funktionen xyz_monitor() im Server-Stub eingeblendet. Mit dieser Funktion können Driver-interne Daten über eine einheitliche Schnittstelle ausgegeben werden. Die konkrete Ausprägung bleibt dem Entwickler des Drivers überlassen.
Ist das Flag nicht gesetzt, kennt der Driver diese Funktionen nicht und kehrt beim Aufruf dieser Funktionen mit dem Fehlercode CMS_FUNCTION zurück.

3.3. Weitergehende Informationen

3.3.1. Start von CMS und von CMS-Applikationen

CMS setzt einen externen **Program Loader** (Monitor, MS/DOS, triviales Startprogramm, ...) voraus. Alle Programme - einschließlich CMS - werden von diesem Program Loader gestartet. Zuerst wird CMS gestartet, initialisiert sich und installiert seine Einsprünge (CMS-Trap, Timer-Interrupt) und kehrt zum Program Loader zurück.

Dieser Loader erscheint für CMS als Prozeß Nr. 1 (s.u.).

Über den Program Loader werden nun die Applikationen gestartet. Diese melden sich bei CMS an (p_create() bzw. d_create()) und kehren ebenfalls wieder zum Loader zurück.

Als letztes wird ein Programm gestartet, das nur eine Endlos-Schleife mit p_schedule() enthält. Dieses Programm braucht sich nicht bei CMS anzumelden, es ist der Idle-Prozeß (s.u.).

Auf ein solches Programm kann auch verzichtet werden, dann dient der Program Loader selbst als Idle-Prozeß. In diesem Fall muß das Time Slicing eingeschaltet sein (mit s_mode() einschalten), damit CMS die Steuerung erhalten kann.

3.3.2. Idle-Prozeß

Der Prozeß mit der Nummer 1 spielt eine Sonderrolle ("Pseudo Prozeß"). Er wird von CMS (unter dem Namen 'system') angelegt und kann nicht gelöscht werden. Er dient sowohl als "Muster" für die Erzeugung weiterer Prozesse als auch als sogenannter Idle-Prozeß.

Jedes Programm startet als Prozeß Nr. 1 ('system'), und das bleibt so bis zum Aufruf von p_create(). In p_create() wird der Kontext des aktuellen Prozesses (also in diesem Fall von Prozeß Nr. 1) in den Kontext des neuen Prozesses kopiert (Muster). Im Kontext des neuen Prozesses werden einige Änderungen vorgenommen (PC, SP), ein Teil des ursprünglichen Stacks wird kopiert. Dann laufen beide Prozesse weiter.

Falls einmal alle Prozesse (einschließlich Prozeß Nr. 1) warten, wird für Prozeß Nr. 1 (Idle Prozeß) der Wartezustand beendet und die entsprechende CMS-Funktion kehrt mit dem Returncode CMS_TIMEOUT zurück. Im Prozeß Nr. 1 sollten also möglichst keine wartenden Aufrufe getätigt werden (bzw. man rechne mit dem vorzeitigen Abbruch des Wartens mit CMS_TIMEOUT).

Die Priorität des Idle-Prozesses sollte auf 0 gesetzt werden, damit er nur aktiv wird, wenn alle anderen Prozesse warten.

3.3.3. Prozeß-Prioritäten

Die Prioritäten der Prozesse liegen zwischen 0 (=CMS_MINPRIO) und CMS_MAXPRIO. Eine größere Prioritätszahl entspricht einer größeren Priorität. CMS_MINPRIO ist für den Idle-Prozeß vorgesehen. Im Bereich von CMS_MINVARPRIO bis CMS_MAXVARPRIO werden die Prioritäten durch CMS dynamisch verändert. Es gilt:

0 = CMS_MINPRIO <= CMS_MINVARPRIO <= CMS_MAXVARPRIO <= CMS_MAXPRIO.

Ansonsten sollen keine Annahmen über die Größenwerte der Prioritäten gemacht werden.

Jedem Prozeß wird zunächst die bei seinem Start vorgegebene Priorität zugeordnet CMS_MINPRIO <= prio <= CMS_MAXPRIO.
Sie kann durch p_priority() jederzeit geändert werden. Diese Priorität entscheidet darüber, wie oft und wie lange ein Prozeß im Vergleich zu den anderen Prozessen die CPU erhält.

3.3.4. Prozeß-Zustände und Scheduling

Mit Scheduling wird die Verteilung der CPU-Zeit auf die einzelnen Prozesse bezeichnet. Die Prozeßwechsel können unter CMS sowohl zeitscheibengesteuert als auch kooperativ erfolgen; dies läßt sich mit der Funktion s_mode() zur Laufzeit einstellen.

Ist kooperatives Scheduling eingestellt, wird nur beim Aufruf einer CMS-Funktion geprüft, ob ein Prozeßwechsel stattfinden kann oder muß.

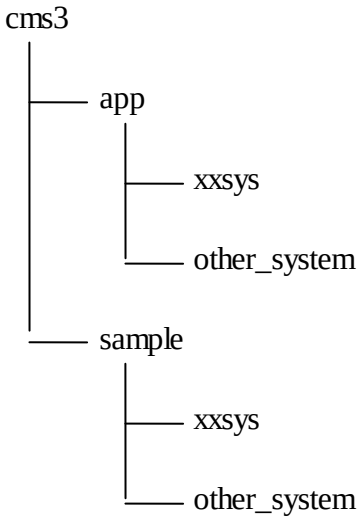
Jeder am Scheduling teilnehmende Prozeß in CMS befindet sich immer in einem der drei Zustände RUN, READY oder WAIT. Der gerade laufende Prozeß hat den Zustand RUN. Die unterbrochenen Prozesse, die aber jederzeit weiterlaufen könnten, haben den READY-Zustand. Bestimmte CMS-Funktionen (zum Beispiel q_read() - Lesen aus einer Queue) können den Prozeß in den Zustand WAIT zu versetzen und damit suspendieren (im obigen Beispiel: wenn die Queue leer ist.). Suspendierte Prozesse können erst wieder starten, wenn eine bestimmte Bedingung erfüllt ist.

Bei allen entsprechenden Funktionen ist eine Maximalzeit ("time out") anzugeben. Nach Ablauf dieser Zeit wird die Prozeß wieder in den Zustand READY versetzt; die gerufene CMS-Funktion kehrt dann mit dem Returncode CMS_TIMEOUT zurück.

Damit werden "dead locks" - mehrere Prozesse warten gleichzeitig aufeinander - aufgelöst. Die Nutzer müssen den Returncode “CMS_TIMEOUT” abtesten und eine sinnvolle Reaktion dafür festlegen.

4. Mitgelieferte Dateien

Folgende Verzeichnisstruktur wird vorgeschlagen:



xxsys steht für das jeweils aktuelle System.

In **app** stehen die systemunabhängigen CMS-Bestandteile, die für Applikationsprogramme unter CMS benötigt werden:

cms3\ app \ drvapi.c
cms3\ app \ drvcend.c
cms3\ app \ drvmain.c
cms3\ app \ cms.h
cms3\ app \ drvapi.h

Die Beispielprogramme, auf die im Text Bezug genommen wurde, befinden sich im Verzeichnis **sample**

cms3\ sample \ d2.c
cms3\ sample \ d2api.c
cms3\ sample \ d2test.c
cms3\ sample \ d2.h
cms3\ sample \ d2api.h
cms3\ sample \ p2.c

Im untergeordneten Verzeichnis **app\xxsys** befinden sich die CPU-abhängigen Bestandteile von CMS und die ausführbaren Dateien von CMS:

cms3\ app \ xxsys \ cmstypes.h
cms3\ app \ xxsys \ CMA.OBJ
cms3\ app \ xxsys \ CMS.EXE

In **sample\xxsys** befinden sich die ausführbaren Dateien der Beispielprogramme:
cms3\ sample \ xxsys \ d2.exe
cms3\ sample \ xxsys \ d2test.exe
cms3\ sample \ xxsys \ p2.exe

Die Namen der Programme hängen vom System ab und können statt *.exe auch *.s oder *.68k usw. lauten.

Teil 2: Referenz

5. Typen

Zur Vereinfachung der Portierung sowohl von CMS als auch von CMS-Applikationen wurden einige Datentypen CMS-weit definiert. Es werden folgende Fälle unterschieden:

- Einfache Daten, bei denen die interne Darstellung "egal" ist, zum Beispiel ein Parameter der Bedeutung Pufferlänge. Hier werden jeweils die Standard-C-Typen verwendet, die in jedem Fall die Mindestbedingungen für die entsprechenden Daten erfüllen, im Beispiel **int**.

- Einfache Daten, bei denen eine bestimmte interne Darstellung vorausgesetzt wird. Für solche Daten werden Typen definiert, die in CMS verwendet werden und auch in den Applikationen zu verwenden sind.

- Funktionstypen. Für Funktionen, die als Parameter von CMS-Aufrufen auftreten können, werden zur Vereinfachnung der Handhabung Typen definiert.

- Strukturierte Typen. Die internen Informationen zu CMS und den aktuellen CMS-Objekten können über strukturierte Typen abgefragt werden (**CMS_**x**CB**). Diese Strukturen und die entsprechenden Abfragefunktionen (x_getcb()) sollen nur für Testzwecke benutzt werden.
  Die Strukturen können sich mit der CMS-Version ändern, es sollte also zuerst diese Version abgefragt werden. Dazu wird festgelegt, daß die SCB-Struktur (mit s_getcb() abfragen) mit der Versionskennung beginnt:

```
typedef
{
    BYTE major;         /* CMS major version number */
    BYTE minor;         /* CMS minor version number */
    ...
} CMS_SCB;
```

Von CMS 3 definierte Typen:

Typname	**Bedeutung**	**Bemerkung**
CHAR/UCHAR/BYTE	1 Byte (8 Bits)	CHAR: signed, UCHAR=BYTE: unsigned
SHORT/USHORT	2 Bytes	SHORT: signed, USHORT: unsigned
LONG/ULONG	4 Bytes	LONG: signed, ULONG: unsigned
STACK	Grundtyp des (Parameter-) Stacks	u.a. zum Zugriff auf Parameter in Drivern
CMS_RETURN	Returncode von CMS-Calls	Integer-Typ (0=CMS_OK, <0: Errorcode)
CMS_HANDLE	Handle von CMS-Objekten	Integer-Typ, sowohl für Handle als auch für Errorcode (Handle > 0, Errorcode < 0)

Typname	Bedeutung	Bemerkung
CMS_TIME	Systemzeit in ms	mindestens 32 Bit
CMS_SIGNAL	Signal, Signalmaske	32 Bit
CMS_ENTRY	Prozeß- und Timeout-Funktion	
CMS_DRVENTRY	Driver-Entry-Funktion	
CMS_SCB	System-Control-Block	s.o.
CMS_xCB	x-Control-Block	für jedes CMS-Objekt z.B. CMS_PCB als Prozeß-Control-Block
CMS_CONTROL	Definition von Umlenkungsbedingungen	siehe q_control()

6. µ-Kernel

6.1. System-Funktionen

6.1.1. s_time() - System-Uhr abfragen

Syntax:

CMS_RETURN s_time(CMS_TIME * time)

Parameter:

CMS_TIME *	time	Systemzeit in msec	OUT

Return:

int	CMS_OK	Funktion erfolgreich beendet

Beschreibung:

Die Funktion s_time() gibt die seit dem Start von CMS vergangene Zeit in Millisekunden zurück. Die Zeit kann mit Hilfe von s_settime() geändert werden. Die Auflösung ist implementierungsabhängig.

6.1.2. s_settime() - System-Uhr setzen

Syntax:

CMS_RETURN s_settime(CMS_TIME time)

Parameter:

CMS_TIME	time	Systemzeit in msec	IN

Return:

int	CMS_OK	Funktion erfolgreich beendet

Beschreibung:

Diese Funktion setzt die Systemzeit von CMS.

6.1.3. s_mode() - CMS Modus ändern

Syntax:

int s_mode(int mode)

Parameter:

int	mode	Betriebsart	IN

Return:

int	CMS_OK	Funktion erfolgreich beendet
	sonst	der bisherige Modus

Beschreibung:

Mit s_mode() wird die CMS-Betriebsart eingestellt, also angegeben, ob das System preemptiv im Zeitscheiben-Modus laufen soll (mode == TIMESLICE setzen) oder kooperativ arbeiten soll (TIMESLICE zurücksetzen). Bei der kooperativen Betriebsart wird bei jedem Aufruf einer CMS-Funktion geprüft, ob ein Prozeßwechsel stattfinden soll.

6.1.4. s_mode2() - CMS Modus ändern – nur in VCMS

Syntax:

void s_mode2(int flags, char * filename)

Parameter:

int	flags	Ausgabeflags	IN
char *	filename	Name des Ausgabefiles	IN

Return:

kein

Beschreibung:

Mit s_mode2() kann in VCMS die Ausgabe von u_out(), u_puts() sowie der .._list()-Funktionen eingestellt werden: Mittels flags werden die Ausgabeziele ausgewählt, falls 'File' gewählt wird, dann kann mit filename der name des Files angegeben werden. Falls filename == NULL ist, dann wird 'debug.out' als Filename verwendet.

Es gelten folgende Werte für flags:

1 Ausgabe an stdout
2 Ausgabe an ein file
4 Ausgabe an das VCMS-window
8 Hiermit kann der Inhalt des VCMS-Windows gelöscht werden.

Die Werte können logisch ODER-verknüpft werden.

6.1.5. s_getcb() - System Control Block abfragen

Syntax:

CMS_RETURN s_getcb(CMS_SCB ** scb)

Parameter:

CMS_SCB **	scb	Pointer auf den Pointer des System-Kontrollblocks	OUT

Return:

int	CMS_OK	Funktion erfolgreich beendet

Beschreibung:

Die Adresse des System-Kontrollblocks von CMS wird in ***scb*** abgelegt, sodaß der Aufrufer die Daten in einem selbstgewählten Format ausgeben kann. Der Kontrollblock darf nicht verändert werden. Er ist versionsabhängig und hat für Version 3.0 folgenden Aufbau:

```
typedef struct                          /* system information block  */
{
   BYTE major;                          /* major version             */
   BYTE minor;                          /* minor version             */
   USHORT clocktick;                    /* tick factor               */
   unsigned long build;                 /* detailed version number   */
   unsigned long cpu;                   /* CPU number                */
   unsigned long syscall_cnt;           /* syscalls (including retries) */
   unsigned long tick_cnt;              /* ticks since cms started   */
   unsigned long schedule_cnt;          /* schedules                 */
   CMS_TIME clock;                      /* clock tick counter        */
   CMS_TIME lost;                       /* (for CMS) lost ticks      */
   char *hardware;                      /* hardware description      */
   BYTE *rambottom;                     /* first RAM address         */
   BYTE *ramtop;                        /* last RAM address          */
   int mode;                            /* system mode               */
} SCB;
```

In allen Versionen steht am Anfang immer die Versionsnummer.

6.1.6. s_list() - System-Informationen ausgeben

Syntax:

void s_list(void)

Parameter:

keine

Return:

kein

Beschreibung:

Es werden Systeminformationen ausgegeben. Für Version 3.0 lautet die Ausgabe beispielsweise

```
SYSTEM LIST
mode         :      1
elapsed time :   6050
system calls :     17
timer ticks  :     48
schedules    :     16
```

Im einzelnen bedeuten die Angaben:

mode	- Betriebsart. Hier preemptiv
elapsed time	- Millisekunden seit CMS-Start
system calls	- Anzahl der Aufrufe von CMS-Funktionen seit Start
timer ticks	- Anzahl der Ticks seit Systemstart
schedules	- Anzahl der Prozeßwechsel seit Systemsart

6.2. Prozeß-Funktionen

6.2.1. p_create() - Prozeß erzeugen

Syntax:

CMS_HANDLE p_create(char * name, CMS_ENTRY entry, int prio, long val, int stcksize)

Parameter:

char *	name	Name des Prozesses	IN
CMS_ENTRY	entry	Adresse der Einsprungfunktion	IN
int	prio	Priorität (0 .. CMS_MAXPRIO)	IN
long	val	Parameter für Einsprungfunktion	IN
int	stcksize	Größe des Stacks	IN

Return:

CMS_HANDLE	> 0	Prozeß-Handle
	CMS_EXIST	ein Prozeß mit diesem Namen existiert bereits
	CMS_FULL	Prozeß-Tabelle voll
	CMS_NOMEM	zuwenig Speicher

Beschreibung:

Es wird ein neuer Prozeß mit dem Namen ***name*** und der Priorität ***prio*** erzeugt. Der Prozeß erhält einen neuen Stack der Größe ***stcksize***. Der Start des Prozesses erfolgt erst mit einem nachfolgenden Aufruf von p_start(). Für den Prozeß erscheint der Start so, als ob die Funktion ***entry*** mit dem Parameter ***val*** aufgerufen worden wäre. p_create() gibt das Prozeß-Handle (>0) oder einen Errorcode (<0) zurück.

Die Funktion ***entry*** darf nicht durch return beendet werden!

Insgesamt können CMS_MAXPROC Prozesse kreiert werden (Konstante in cms.h).

6.2.2. p_delete() - Prozeß löschen

Syntax:

CMS_RETURN p_delete(CMS_HANDLE phandle)

Parameter:

CMS_HANDLE	phandle	Handle des Prozesses	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Der mit ***phandle*** identifizierte Prozeß wird gelöscht, d.h. sein Eintrag in der Prozeßtabelle von CMS wird gelöscht, und er nimmt nicht mehr am Scheduling teil.

6.2.3. p_open() - Prozeß öffnen

Syntax:

CMS_HANDLE p_open(char * name)

Parameter:

char *	name	Name des Prozesses	IN

Return:

CMS_HANDLE	> 0	Handle des Prozesses
	CMS_NOTFND	Prozeß nicht gefunden

Beschreibung:

Mit p_open() wird ein Prozeß geöffnet. Dazu wird der Name des zu öffnenden Prozesses angegeben. Bei Erfolg meldet CMS das Prozeß-Handle zurück, welches bei allen nachfolgenden Zugriffen auf den Prozeß benutzt wird.

6.2.4. p_close() - Prozeß schließen

Syntax:

CMS_RETURN p_close(CMS_HANDLE phandle)

Parameter:

CMS_HANDLE	phandle	Handle des Prozesses	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit p_close() wird der Zugriff auf einen Prozeß beendet.

6.2.5. p_handle() - Nächstes Prozeß-Handle liefern

Syntax:

CMS_HANDLE p_handle(CMS_HANDLE phandle)

Parameter:

CMS_HANDLE	phandle	Handle des Prozesses	IN

Return:

CMS_ HANDLE	> 0	Handle des nächsten Prozesses
	= 0	keine weiteren Handles vorhanden

Beschreibung:

Diese Funktion liefert zu einem gültigen Prozeß-Handle das nächste gültige Prozeß-Handle. Für ***phandle*** = 0 wird das erste Prozeß-Handle geliefert, für das letzte gültige Prozeß-Handle wird 0 zurückgeliefert.

6.2.6. p_start() - Prozeß starten

Syntax:

CMS_RETURN p_start(CMS_HANDLE phandle)

Parameter:

CMS_HANDLE	phandle	Handle des Prozesses	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit p_start() wird ein Prozeß gestartet, der sich im Zustand STOP befindet. Dies ist der Fall, nachdem eine Prozeß angelegt (p_create()) bzw. angehalten (p_stop()) wurde. Nach dem Starten nimmt der Prozeß (wieder) am Scheduling teil.

6.2.7. p_stop() - Prozeß anhalten

Syntax:

CMS_RETURN p_stop(CMS_HANDLE phandle)

Parameter:

CMS_HANDLE	phandle	Handle des Prozesses	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit p_stop() wird ein Prozeß angehalten. Er nimmt dann solange nicht mehr am Scheduling teil, bis er durch den Aufruf von p_start() wieder gestartet wird.

6.2.8. p_enable() - Scheduling einschalten

Syntax:

void p_enable(void)

Parameter:

kein

Return:

kein

Beschreibung:

Die Funktion p_enable() läßt Prozeßwechsel und Aufrufe von Timeout-Funktionen wieder zu, nachdem sie mit p_disable() gesperrt waren.

6.2.9. p_disable() - Scheduling ausschalten

Syntax:

void p_disable(void)

Parameter:

keine

Return:

kein

Beschreibung:

Die Funktion p_disable() sperrt alle Prozeßwechsel sowie den Aufruf von Timeout-Funktionen. Damit ist gesichert, daß die folgenden Anweisungen ohne Unterbrechung abgearbeitet werden. Diese Funktion kann zum Beispiel genutzt werden, um kritische Daten zu bearbeiten, die von einer Timeout-Funktion verändert werden könnten. Nach Bearbeitung der kritischen Anweisungen muß der Prozeß wieder p_enable() aufrufen.

6.2.10. p_priority() - Prozeßpriorität ändern

Syntax:

int p_priority(CMS_HANDLE phandle, int prio)

Parameter:

CMS_HANDLE	phandle	Handle des Prozesses	IN
int	prio	neue Priorität des Prozesses	IN

Return:

int	>= 0	alte Priorität des Prozesses
	CMS_BADHANDLE	ungültiges Handle

Beschreibung:

Mit p_priority() kann die Priorität eines Prozesses dynamisch geändert werden.

Die Priorität ist ein positiver Wert zwischen 0 und CMS_MAXPRIO, wobei ein hoher Wert eine hohe Priorität repräsentiert.

6.2.11. p_schedule() - CPU explizit abgeben

Syntax:

void p_schedule(void)

Parameter:

kein

Return:

kein

Beschreibung:

Mit p_schedule() kann ein Prozeß die CPU explizit abgeben. Der Prozeß wird dadurch in den Zustand READY (to run) gesetzt, und es erfolgt ein Schedule.

6.2.12. p_pid() - Eigenes Prozeßhandle abfragen

Syntax:

CMS_HANDLE p_pid(void)

Parameter:

keine

Return:

CMS_HANDLE	phandle	Handle des Prozesses

Beschreibung:

Mit p_pid() erfährt der Prozeß sein eigenes Handle. Es wird zum Beispiel benötigt, wenn er sich selbst bei CMS abmelden will:

```
p_delete ( p_pid() );
```

6.2.13. p_sleep() - delay msec warten

Syntax:

CMS_RETURN p_sleep(CMS_TIME delay)

Parameter:

CMS_TIME	delay	Wartezeit in Millisekunden	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet

Beschreibung:

Die aktuelle Prozeß wird für ***delay*** Millisekunden suspendiert.

6.2.14. p_wait() - Auf Signal warten

Syntax:

CMS_RETURN p_wait(CMS_SIGNAL mask, CMS_TIME timeout)

Parameter:

CMS_SIGNAL	mask	Signalmaske	IN
CMS_TIME	timeout	Wartezeit in Millisekunden	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_ERROR	unzulässige Signalmaske
	CMS_TIMEOUT	Wartezeit abgelaufen

Beschreibung:

Der rufende Prozeß wird solange suspendiert, bis er ein Signal aus ***mask*** erhalten hat (d.h. bis Signalwort & ***mask*** != 0) bzw. die Zeit ***timeout*** (in ms) abgelaufen ist.

6.2.15. p_send() - Signal senden

Syntax:

CMS_RETURN p_send(CMS_HANDLE phandle, CMS_SIGNAL signal)

Parameter:

CMS_HANDLE	phandle	Prozeß-Handle	IN
CMS_SIGNAL	signal	Signalwort	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	ungültiges Handle

Beschreibung:

Diese Funktion sendet das Signal ***signal*** an den Prozeß ***phandle***.
Drei Signale und ihre Positionen sind bereits von CMS definiert , eine Postion ist reserviert. Sie sollen nicht für private Signalisierungen der Prozesse verwendet werden:

```
#define SIG_RETRY    0x80000000L       /* retry CMS call            */
#define SIG_TIMEOUT  0x40000000L       /* timeout of CMS call       */
#define SIG_TIMER    0x20000000L       /* timer intervall elapsed   */
                     0x10000000L       /* reserve   */
```

6.2.16. p_getset() - Eigenes Signalwort abfragen und neu setzen

Syntax:

CMS_RETURN p_getset(CMS_SIGNAL * oldval, CMS_SIGNAL newval,
CMS_SIGNAL mask)

Parameter:

CMS_SIGNAL *	oldval	Altes Signalwort	OUT
CMS_SIGNAL	newval	neuese Signalwort	IN
CMS_SIGNAL	mask	Signalmaske	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet

Beschreibung:

Mit Hilfe dieser Funktion kann ein Prozeß sein eigenes Signalwort abfragen. Der Wert wird über ***oldval*** zurückgegeben. Gleichzeitig wird das Signalwort mit dem Wert ***newval*** ganz oder teilweise neu gesetzt: Es werden die Bits im Signalwort neu mit den entsprechenden Bits aus ***newval*** gesetzt, die in ***mask*** 1 sind (d.h. Signalwort := (Signalwort & ~***mask***) | (***newval*** & ***mask***)).

6.2.17. p_getcb() - Prozeß Control Block abfragen

Syntax:

CMS_RETURN p_getcb(CMS_HANDLE phandle, CMS_PCB ** pcb)

Parameter:

CMS_HANDLE	phandle	Prozeß-Handle	IN
CMS_PCB **	pcb	Adresse des Kontrollblocks	OUT

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	ungültiges Handle

Beschreibung:

Es wird ein Zeiger auf den Prozeß-Kontrollblock von ***phandle*** zurückgegeben. Der Kontrollblock enthält die aktuellen charakteristischen Daten eines Prozesses und darf nicht verändert werden !
In Version 3.0 hat er folgenden Aufbau:

```
typedef struct                          /* process control block     */
{
   CMS_SIGNAL signal;                   /* process signal            */
   CMS_SIGNAL sigmask;                  /* signal mask               */
   BYTE * stack;                        /* process stack area        */
   void * initct;                       /* init time context pointer */
   CMS_HANDLE timer;                    /* timer handle              */
   unsigned int opencnt;                /* open counter              */
   char name[NAMELEN];                  /* process name              */
   BYTE prio_start;                     /* process priority at start */
   BYTE prio_act;                       /* actual process priority   */
   BYTE state;                          /* process state             */
   BYTE dummy;
}  PCB;
```

6.2.18. p_list() - Prozeß-Informationen ausgeben

Syntax:

void p_list(void)

Parameter:

keine

Return:

kein

Beschreibung:

Es werden Prozeß-Informationen zu allen angemeldeten Prozessen ausgegeben. Die Liste hat folgende Form:

```
PROCESS LIST:
pid name      stack   prio state  signal    sigmask
  1 system   $00000000   3   2   $00000000 $00000000
  2 proc1    $22D5E3EE   4   3   $00000000 $20000000
  3 proc2    $22D5D41E  68   3   $00000000 $20000000
  4 proc3    $22D5C44E  69   1   $00000000 $00000000
```

Im einzelnen bedeuten die Parameter:

pid	Prozeß-Handle
name	Name des Prozesses
stack	Adresse des Stack-Bereichs (“system” hat immer die Stack-Adresse 0)
prio	Priorität des Prozesses
state	Status des Prozesses: RUN=1; READY=2; WAIT=3; STOPPED_READY=4; STOPPED_WAIT=5
signal	aktueller Wert des Signalwortes des Prozesses
sigmask	gesetzte Signalmaske (für p_wait())

6.3. Timer-Funktionen

6.3.1. t_create() - Timer anlegen

Syntax:

CMS_HANDLE t_create(char * name)

Parameter:

char *	name	Name des Timers	IN

Return:

CMS_HANDLE	> 0	Timer-Handle
	CMS_EXIST	Timer existiert schon
	CMS_FULL	Timer-Tabelle ist voll
	CMS_NOMEM	zuwenig Speicher frei

Beschreibung:

Mit t_create() wird ein Timer angelegt. Dies bedeutet, daß CMS in der Timer-Tabelle einen Eintrag für diesen Timer anlegt. Insgesamt können CMS_MAXTIMER Timer kreiert werden (Konstante in cms.h).

6.3.2. t_delete() - Timer löschen

Syntax:

CMS_RETURN t_delete(CMS_HANDLE thandle)

Parameter:

CMS_HANDLE	handle	Timer-Handle

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	ungültiges Handle

Beschreibung:

Mit t_delete() wird ein Timer gelöscht. Dies bedeutet, daß CMS in der Timer-Tabelle den Eintrag für diesen Timer löscht. Weitere Zugriffe auf den Timer sind nicht möglich.

Über einen Zähler wird geprüft, ob der zu löschende Timer noch von anderen Prozessen benutzt wird (noch geöffnet ist). Ist dies der Fall, wird der Timer nicht gelöscht, sondern zum Löschen vorgemerkt und nach dem letzten t_close() gelöscht.

6.3.3. t_open() - Timer öffnen

Syntax:

CMS_HANDLE t_open(char * name)

Parameter:

char *	name	Name des Timers	IN

Return:

CMS_HANDLE	CMS_OK	Timer-Handle
	CMS_NOTFND	Timer nicht gefunden

Beschreibung:

Mit t_open() wird ein Timer geöffnet. Dazu wird der Name des zu öffnenden Timers angegeben. Bei Erfolg meldet CMS das Timer-Handle zurück, welches bei allen nachfolgenden Zugriffen auf den Timer benutzt wird.

Bei jedem Open wird ein Zähler für diesen Timer inkrementiert. Er wird beim Versuch, den Timer zu löschen, ausgewertet.

6.3.4. t_close() - Timer schließen

Syntax:

CMS_RETURN t_close(CMS_HANDLE thandle)

Parameter:

CMS_HANDLE	handle	Timer-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	ungültiges Handle

Beschreibung:

Mit t_close() wird der Zugriff auf einen Timer beendet. Ein weiterer Zugriff ist erst nach erneutem Open möglich.

Bei jedem Close auf einen Timer wird sein Zähler dekrementiert. Dieser gibt an, ob der Timer von einem Prozeß noch benötigt wird.

6.3.5. t_handle() - Nächstes Timer-Handle liefern

Syntax:

CMS_HANDLE t_handle(CMS_HANDLE thandle)

Parameter:

CMS_HANDLE	handle	Timer-Handle	IN

Return:

CMS_ HANDLE	> 0	das nächste Timer-Handle
	= 0	kein weiteres Handle vorhanden

Beschreibung:

Diese Funktion liefert zu einem gültigen Timer-Handle das nächste gültige Timer-Handle. Für ***thandle*** = 0 wird das erste Timer-Handle geliefert, für das letzte gültige Timer-Handle wird 0 zurückgeliefert.

6.3.6. t_start() - Timer starten mit Timeout-Funktion

Syntax:

CMS_RETURN t_start(CMS_HANDLE thandle, CMS_TIME time,
CMS_ENTRY tofun, long parm, int periodic)

Parameter:

CMS_HANDLE	handle	Timer-Handle	IN
CMS_TIME	time	Laufzeit des Timers in ms	IN
CMS_ENTRY	tofun	Funktionsadresse	IN
long	parm	Parameter für diese Funktion	IN
int	periodic	Flag für einmaliges / wiederholtes Starten	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	ungültiges Handle

Beschreibung:

Der Timer ***thandle*** wird gestartet, nach ***time*** ms ist er abgelaufen. Nun wird er in Abhängigkeit des ***periodic***-Flags entweder angehalten (***periodic*** = 0) oder sofort automatisch mit denselben Parametern wieder neu gestartet (***periodic*** = 1).

Falls ***tofun*** kein NULL-Pointer ist, wird nach Ablauf des Timers die Timeout-Funktion ***tofun*** mit dem Parameter ***parm*** aufgerufen. Die Timeout-Funktion wird im Kontext des Prozesses aufgerufen, in dem sie sich befindet (d.h. der den Timer gestartet hat); nach Rückkehr aus dieser Funktion geht der Ablauf so weiter, als wäre sie nicht aufgerufen worden. CMS-Aufrufe in der Timeout-Funktion sind erlaubt.

6.3.7. t_stop() - Timer anhalten

Syntax:

CMS_RETURN t_stop(CMS_HANDLE thandle)

Parameter:

CMS_HANDLE	handle	Timer-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	ungültiges Handle

Beschreibung:

Mit t_stop() wird ein Timer vor Ablauf seiner Laufzeit angehalten. Er kann durch den Aufruf von t_start() wieder neu gestartet werden. Die eventuell übergebene Timerfunktion wird nicht ausgeführt.

6.3.8. t_signal() - Timer-Signal definieren

Syntax:

CMS_RETURN t_signal(CMS_HANDLE thandle, int kind, CMS_SIGNAL signal)

Parameter:

CMS_HANDLE	handle	Timer-Handle	IN
int	kind	Art des Ereignisses	IN
CMS_SIGNAL	signal	Signalwert	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	ungültiges Handle

Beschreibung:

Bei Eintreten des mit ***kind*** spezifizierten Ereignisses des Timers ***thandle*** wird an den rufenden Prozeß das Signal ***signal*** gesendet. Dieses Verhalten bleibt für diesen Timer solange bestehen, bis es durch den erneuten Aufruf von t_signal() im selben Prozeß wieder geändert wird.
Es ist im Moment nur ein Timer-Ereignis definiert, das signalisiert werden kann: der Ablauf des Timers; daher wird ***kind*** in der aktuellen Version von CMS ignoriert.

6.3.9. t_status() - Timer-Status abfragen

Syntax:

CMS_RETURN t_status(CMS_HANDLE thandle, CMS_TIME * time)

Parameter:

CMS_HANDLE	handle	Timer-Handle	IN
CMS_TIME *	time	Restlaufzeit des Timers	OUT

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	ungültiges Handle

Beschreibung:

Diese Funktion liefert über ***time*** die Restzeit (in ms) bis zum Ablauf des Timers zurück.

6.3.10. t_wait() - auf Timer warten

Syntax:

CMS_RETURN t_wait(CMS_HANDLE thandle)

Parameter:

CMS_HANDLE	handle	Timer-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	ungültiges Handle

Beschreibung:

Mit t_wait() kann bis zum Timeout eines laufenden Timers gewartet werden; der rufende Prozeß wird bis zu diesem Zeitpunkt suspendiert.

6.3.11. t_getcb() - Timer Control Block abfragen

Syntax:

CMS_RETURN t_getcb(CMS_HANDLE thandle, CMS_TCB ** tcb)

Parameter:

CMS_HANDLE	handle	Timer-Handle

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	ungültiges Handle

Beschreibung:

Es wird ein Zeiger auf den Timer-Kontrollblock von ***thandle*** zurückgegeben. Der Kontrollblock enthält die aktuellen charakteristischen Daten des Timers und darf nicht verändert werden !
Er hat folgenden Aufbau:

```
typedef struct                          /* timer control block       */
{
   CMS_SIGNAL signal;                   /* signal to send            */
   CMS_TIME delay;                      /* timeout delay             */
   CMS_TIME clock;                      /* timeout clock             */
   void (*timeout)();                   /* timeout fun. for starter  */
   long parm;                           /* parameter for timeout fn. */
   CMS_SIGNAL siglist[MAXPROC];         /* processes to be notified  */
   unsigned int opencnt;                /* open counter              */
   BYTE sigflag;                        /* any process to notify ?   */
   BYTE running;                        /* flag: running or stopped  */
   PROC plist[MAXPROC];                 /* list of waiting processes */
   char name[NAMELEN];                  /* timer name                */
   PROC starter;                        /* starter process           */
   BYTE periodic;                       /* flag                      */
} TCB;
```

6.3.12. t_list() - Timer-Informationen ausgeben

Syntax:

void t_list(void)

Parameter:

keine

Return:

kein

Beschreibung:

Es werden Informationen zu allen Timern tabellarisch ausgegeben. Die Liste hat die folgende Form:

```
TIMER LIST:
  hdl name      signal         rest pid  timeout    parm   per
 1000 _P1      $00000000          0   0 $00000000      0     0
 1001 _P2      $20000000          0   2 $00000000      0     0
 1002 _P3      $20000000      12980   3 $00000000      0     0
 1003 _P4      $20000000          0   4 $00000000      0     0
 1004 proc1tim $20000000        550   2 $326201FA     57     0
```

Im einzelnen bedeuten die Parameter:

hdl	Timer Handle
name	Name des Timers
signal	Signal, das an einen wartenden Prozeß zu senden ist
rest	Restlaufzeit
pid	Handle des Prozesse, der bei Timer-Ablauf benachrichtigt werden soll.
timeout	Adresse der Timeout-Funktion
parm	Parameter für Timeout-Funktion
per	Periodoc-Flag

6.4. Utility-Funktionen

6.4.1. u_alloc() - Speicher allozieren

Syntax:

CMS_RETURN u_alloc(void ** addr, unsigned int size)

Parameter:

void **	addr	Feld für die Speicheradresse	OUT
unsigned int	size	Speichergröße in Byte	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_SIZE	unsinniger Längenparameter
	CMS_NOMEM	zuwenig Speicher frei

Beschreibung:

Diese Funktion alloziert ***size*** Bytes Speicher in einem globalen Heap. Über ***addr*** wird ein Pointer auf den allozierten Speicher zurückgegeben. u_alloc() entspricht seiner Funktionalität nach der C-Funktion malloc() und soll statt dieser benutzt werden, da auf diese Weise CMS die Verwaltung des Speichers erhält.

6.4.2. u_free() - Speicher freigeben

Syntax:

CMS_RETURN u_free(void * addr)

Parameter:

void *	addr	Speicheradresse	IN

Return:

CMS_RETURN	CMS_OK	Speicher freigegeben
	CMS_ERROR	addr ist nicht Startadresse eines allokierten Speicherbereichs

Beschreibung:

Diese Funktion gibt den mit u_alloc() allozierten Speicher ab Adresse ***addr*** wieder frei. ***addr*** muß eine von u_alloc() zurückgelieferte Adresse sein und nicht eine beliebige Adresse innerhalb eines allokierten Speicherbereichs. u_free() entspricht der C-Funktion free() und soll statt dieser benutzt werden.

6.4.3. u_out() - String 'raw' ausgeben

Syntax:

int u_out(char * string)

Parameter:

char *	string	Zeichenkette (ASCII, nullterminiert)	IN

Return:

int	>= 0	Anzahl der übertragenen Zeichen (außer der abschließenden Null)

Beschreibung:

Diese Funktion gibt ***string*** uninterpretiert ('roh') an den Bildschirm aus (Was dabei unter 'Bildschirm' zu verstehen ist, hängt von der Implementierung ab.). Zwei Zeichen werden allerding doch interpretiert: Carriage Return ('\r') und Line Feed ('\n'). Es wird die Anzahl der ausgegebenen Zeichen (=Stringlänge) zurückgegeben.

Mit u_route() kann das Ziel der Ausgabe geändert werden.

6.4.4. u_puts() - String 'cooked' ausgeben

Syntax:

int u_puts(char * string)

Parameter:

char *	string	Zeichenkette (ASCII, nullterminiert)	IN

Return:

int	nicht definiert

Beschreibung:

Diese Funktion gibt ***string*** an den Bildschirm aus (Was dabei unter 'Bildschirm' zu verstehen ist, hängt von der Implementierung ab.). Dabei werden bestimmte Zeichen interpretiert (u.a. '\n' als Carriage Return + Line Feed). Der Rückgabewert ist (vorerst) nicht definiert. Die Wirkung entspricht dem Aufruf fputs(***string***,stdout).

Mit u_route() kann das Ziel der Ausgabe geändert werden.

6.4.5. u_route() - CMS-Ausgaben umleiten

Syntax:

CMS_HANDLE u_route(CMS_HANDLE h)

Parameter:

CMS_HANDLE	h	0 oder 1 oder Handle eines Kommunikationsobjektes	IN

Return:

CMS_HANDLE	vorheriger Routing-Wert (wie h) oder Errorcode (< 0)

Beschreibung:

Mit dieser Funktion können die "Bildschirm"-Ausgaben von CMS umgeroutet werden:

h = 0	keine Ausgaben
h = 1	"normale" Ausgaben (Default)
sonst	***h*** muss das CMS-Handle einer Queue, einer Pipe oder eines (IPC-)Drivers sein. Alle Ausgaben werden dann an dieses "Objekt" geroutet.

Die Funktion gibt das vorherige Ausgaberouting zurück.

7. Prozeß Synchronisation

7.1. Semaphore-Funktionen

7.1.1. r_create() - Semaphore anlegen

Syntax:

CMS_HANDLE r_create(char * name)

Parameter:

char *	name	Name der Semaphore	IN

Return:

CMS_HANDLE	> 0	Semaphore-Handle
	CMS_EXIST	Semaphore existiert bereits
	CMS_FULL	Semaphore-Tabelle voll

Beschreibung:

Mit r_create() wird eine Semaphore angelegt. Dies bedeutet, daß CMS in der Semaphoren-Tabelle einen Eintrag für diese Semaphore anlegt und verwaltet.

Durch die Semaphoren kann der konkurrierende Zugriff mehrerer Prozesse auf Resourcen (Speicher, Peripherie, Programme u.ä.) synchronisiert werden. Der aktuelle Zustand einer Semaphore wird von CMS verwaltet und kann mit r_status() abgefragt werden. Der Wert 0 bedeutet, daß die Semaphore frei ist. Eine Zahl > 0 gibt an, wie oft ein Prozeß die Semaphore angefordert hat.

Insgesamt können CMS_MAXRES Semaphoren kreiert werden (Konstante in cms.h).

7.1.2. r_delete() - Semaphore löschen

Syntax:

CMS_RETURN r_delete(CMS_HANDLE rhandle)

Parameter:

CMS_HANDLE	rhandle	Semaphore-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit r_delete() wird die durch ***rhandle*** identifizierte Semaphore gelöscht. Dies bedeutet, daß CMS in der Semaphoren-Tabelle den Eintrag für diese Semaphore löscht. Weitere Zugriffe auf die Semaphore sind nicht möglich.

Über den Open-Zähler der Semaphore wird geprüft, ob sie von anderen Prozessen noch benutzt wird (noch geöffnet ist). Ist dies der Fall, wird die Semaphore nicht gelöscht, sondern zum Löschen vorgemerkt und nach dem letzten r_close() auf die Semaphore gelöscht.

7.1.3. r_open() - Semaphore öffnen

Syntax:

CMS_HANDLE r_open(char * name)

Parameter:

char *	name	Name der Semaphore	IN

Return:

CMS_HANDLE	rhandle	Semaphore-Handle
	CMS_NOTFND	Semaphore nicht gefunden

Beschreibung:

Mit r_open() wird eine Semaphore geöffnet. Dazu wird der Name der zu öffnenden Semaphore angegeben. Bei Erfolg meldet CMS das Semaphore-Handle zurück, welches bei allen nachfolgenden Zugriffen auf die Semaphore benutzt wird.

Bei jedem Open auf eine Semaphore wird ihr Open-Zähler inkrementiert; dieser gibt an, ob sie von einem Prozeß noch benötigt wird. In einer Applikation muß daher für jeden Aufruf von r_open() auch ein r_close() erfolgen, bevor die Applikation beendet wird.

7.1.4. r_close() - Semaphore schließen

Syntax:

CMS_RETURN r_close(CMS_HANDLE rhandle)

Parameter:

CMS_HANDLE	rhandle	Semaphore-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit r_close() wird der Zugriff auf eine Semaphore beendet. Ein weiterer Zugriff ist erst nach erneutem Open möglich.

Bei jedem Close auf eine Semaphore wird ihr Open-Zähler dekrementiert; dieser gibt an, ob sie von einem Prozeß noch benötigt wird. In einer Applikation muß daher für jeden Aufruf von r_open() auch ein r_close() erfolgen, bevor die Applikation beendet wird.

7.1.5. r_handle() - Nächstes Semaphore-Handle liefern

Syntax:

CMS_HANDLE r_handle(CMS_HANDLE rhandle)

Parameter:

CMS_HANDLE	rhandle	Semaphore-Handle	IN

Return:

CMS_HANDLE	> 0	nächstes Semaphore-Handle
	= 0	kein weiteres Handle vorhanden

Beschreibung:

Diese Funktion liefert zu einem gültigen Semaphore-Handle das nächste gültige Semaphore-Handle. Für ***rhandle*** = 0 wird das erste Semaphore-Handle geliefert, für das letzte gültige Semaphore-Handle wird 0 zurückgeliefert.

7.1.6. r_request() - Semaphore anfordern

Syntax:

CMS_RETURN r_request(CMS_HANDLE rhandle, CMS_TIME timeout)

Parameter:

CMS_HANDLE	rhandle	Semaphore-Handle	IN
CMS_TIME	timeout	Wartezeit	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	Ungültiges Handle
	CMS_TIMEOUT	kein Erfolg nach Ablauf der Wartezeit

Beschreibung:

Mit r_request() kann ein Prozeß eine Semaphore zur exklusiven Nutzung anfordern; bei Erfolg hat er solange alleinigen Zugriff auf die Semaphore, bis er sie wieder freigibt (r_release()).

Ist zum Zeitpunkt des Aufrufs die angeforderte Semaphore nicht frei, d.h. von einem anderen Prozeß belegt, wird der rufende Prozeß solange suspendiert, bis ihm die Semaphore zugeordnet werden kann oder die durch Timeout angegebene Zeit abläuft. Im letzten Fall liefert die Funktion den Returncode CMS_TIMEOUT zurück.

Derselbe Prozeß kann eine Semaphore mehrmals anfordern; dann wird ein Request-Zähler incrementiert. Er muß sie ebensooft freigeben.

Für jeden Aufruf von r_request() muß auch ein Aufruf von r_release erfolgen.

7.1.7. r_release() - Semaphore freigeben

Syntax:

CMS_RETURN r_release(CMS_HANDLE rhandle)

Parameter:

CMS_HANDLE	rhandle	Semaphore-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit r_release() wird eine zur exklusiven Nutzung zugeordnete Semaphore wieder freigegeben. Danach haben andere Prozesse wieder die Möglichkeit, auf die betreffende Semaphore zuzugreifen (sofern der freibegende Prozeß diese Semaphore nicht mehrfach angefordert hat. Dann muß er sie ebensooft freigeben.)

7.1.8. r_signal() - Semaphore-Signal definieren

Syntax:

CMS_RETURN r_signal(CMS_HANDLE rhandle, int kind, CMS_SIGNAL signal)

Parameter:

CMS_HANDLE	rhandle	Semaphore-Handle	IN
int	kind	Ereignisart	IN
CMS_SIGNAL	signal	Signalewort	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Bei Eintreten des mit ***kind*** spezifizierten Ereignisses der Semaphore ***rhandle*** wird an den rufenden Prozeß das Signal ***signal*** gesendet. Dieses Verhalten bleibt für diese Semaphore solange bestehen, bis es durch erneuten Aufruf von r_signal() im selben Prozeß wieder geändert wird.
Es ist im Moment nur ein Semaphore-Ereignis definiert, das signalisiert werden kann: das Freiwerden der Semaphore.

7.1.9. r_status() - Semaphore-Status abfragen

Syntax:

int r_status(CMS_HANDLE rhandle)

Parameter:

CMS_HANDLE	rhandle	Semaphore-Handle	IN

Return:

CMS_RETURN	>= 0	Anzahl der Requests auf diese Semaphore
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Bei gültigem Semaphore-Handle ***handle*** wird die Anzahl der Requests auf diese Semaphore zurückgegeben, also 0, wenn sie frei ist, oder ein positive ganze Zahl, wenn ein Prozeß sie ein- oder mehrmals angefordert hat.

7.1.10. r_getcb() - Semaphore Control Block abfragen

Syntax:

CMS_RETURN r_getcb(CMS_HANDLE rhandle, CMS_RCB ** rcb)

Parameter:

CMS_HANDLE	rhandle	Semaphore-Handle	IN
CMS_RCB **	rcb	Feld für die Adresse des Kontrollblocks	OUT

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich beendet
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Es wird ein Zeiger auf den Semaphore-Kontrollblock von ***rhandle*** zurückgegeben. Der Kontrollblock darf nicht verändert werden !
Er hat folgenden Aufbau:

```
typedef struct                          /* semaphore control block   */
{
   CMS_SIGNAL siglist[CMS_MAXPROC];     /* corresponding to nlist    */
   unsigned int opencnt;                /* open counter              */
   unsigned int req;                    /* request counter           */
   int delreq;                          /* delete requested          */
   PROC holder;                         /* semaphore holder          */
   PROC plist[CMS_MAXPROC];             /* list of waiting processes */
                                        /*    (for timeout / retry)  */
                                        /*    plist[0] is the owner  */
   PROC nlist[CMS_MAXPROC];             /* processes to be notified  */
   char name[NAMELEN];                  /* semaphore name            */
} RCB;
```

7.1.11. r_list() - Semaphore-Informationen ausgeben

Syntax:

void r_list(void)

Parameter:

keine

Return:

kein

Beschreibung:

Es wird eine Tabelle mit Informationen zu allen Semaphoren ausgegeben. Sie hat folgenden Aufbau:

```
SEMAPHORE LIST:
  h name     value    count
```

Im einzelnen bedeuten die Parameter:

h	Semaphore-Handle
name	Name der Semaphore
value	Anzahl der Request auf die Semaphore
count	Anzahl der Benutzer der Semaphore (Open-Zähler)

7.2. Event-Funktionen

7.2.1. e_create() - Event anlegen

Syntax:

CMS_HANDLE e_create(char * name)

Parameter:

char *	name	Name des Events

Return:

CMS_HANDLE	ehandle	Event-Handle
	CMS_EXIST	Event existiert schon
	CMS_FULL	Event-Tabelle ist voll

Beschreibung:

Mit dieser Funktion wird ein neues Event mit dem Namen ***name*** angelegt. Das Event ist eine 16-Bit-Variable, die allen Prozessen zugänglich ist und auf der bestimmte arithmetische und Bit-Operationen ausgeführt werden können. Auf ein Event kann gewartet werden, d.h. ein Prozeß wird solange suspendiert, bis das Event eine bestimmte Bedingung erfüllt.

Es können maximal CMS_MAXEVENTS Events angelegt werden (Konstante in CMS.H).

7.2.2. e_delete() - Event löschen

Syntax:

CMS_RETURN e_delete(CMS_HANDLE ehandle)

Parameter:

CMS_HANDLE	ehandle	Event-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit e_delete() wird ein Event gelöscht. Dies bedeutet, daß CMS in der Event-Tabelle den Eintrag für dieses Event löscht; weitere Zugriffe auf das Event sind nicht möglich.

Über einen Open-Zähler wird geprüft, ob das zu löschende Event von anderen Prozessen noch benutzt wird (noch geöffnet ist). Ist dies der Fall, wird das Event nicht gelöscht, sondern zum Löschen vorgemerkt und nach dem letzten close auf dieses Event gelöscht.

7.2.3. e_open() - Event öffnen

Syntax:

CMS_HANDLE e_open(char * name)

Parameter:

char *	name	Name des Events	IN

Return:

CMS_HANDLE	ehandle	Event-Handle
	CMS_NOTFND	Event nicht gefunden

Beschreibung:

Mit e_open() wird ein Event geöffnet. Dazu wird der Name des zu öffnenden Event angegeben. Bei Erfolg meldet CMS das Event-Handle zurück, welches bei allen nachfolgenden Zugriffen auf das Event benutzt wird.

Bei jedem Open wird der Zähler des Events inkrementiert. Dieser gibt an, wieviele Benutzer ein Event hat.

7.2.4. e_close() - Event schließen

Syntax:

CMS_RETURN e_close(CMS_HANDLE ehandle)

Parameter:

CMS_HANDLE	ehandle	Event-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit e_close() wird der Zugriff auf ein Event beendet. Ein weiterer Zugriff ist erst nach erneutem e_open() möglich.

Bei jedem Close wird der Zähler des Events dekrementiert. Dieser gibt an, ob ein CMS-Objekt von einem Prozeß noch benötigt wird.

7.2.5. e_handle() - Nächstes Event-Handle liefern

Syntax:

CMS_HANDLE e_handle(CMS_HANDLE ehandle)

Parameter:

CMS_HANDLE	ehandle	Event-Handle	IN

Return:

CMS_HANDLE	> 0	nächstes Event-Handle
	= 0	kein weiteres Handle vorhanden

Beschreibung:

Diese Funktion liefert zu einem gültigen Event-Handle das nächste gültige Event-Handle. Für ***ehandle*** = 0 wird das erste Event-Handle geliefert, für das letzte gültige Event-Handle wird 0 zurückgeliefert.

7.2.6. e_set() - Event setzen

Syntax:

CMS_RETURN e_set(CMS_HANDLE ehandle, int op, int value)

Parameter:

CMS_HANDLE	ehandle	Event-Handle	IN
int	op	Operation	IN
int	value	Wert	

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit e_set() wird ein Event gesetzt. Dazu werden ein Operationscode ***op*** und ein Wert ***value*** spezifiziert. Der interne Wert des Events wird mit dem Parameter ***value*** verknüpft, was auf den Wert des Events folgende Auswirkung hat:

Opcode	Bedeutung
'='	event = value
'+'	event += value
'-'	event -= value
'|'	event |= value (bitweises OR)
'&'	event &= value (bitweises AND)
'^'	event ^= value (bitweises XOR).

Mit diesen Operationen ist es möglich, ein Event wie einen Satz von Signalen oder auch als Counter zu organisieren; auch Mischformen sind möglich.

7.2.7. e_wait() - Auf Event warten

Syntax:

CMS_RETURN e_wait(CMS_HANDLE ehandle, int op, int value, CMS_TIME timeout)

Parameter:

CMS_HANDLE	ehandle	Event-Handle	IN
int	op	Vergleichsoperation	IN
int	value	Vergleichswert	IN
CMS_TIME	timeout	Wartezeit in Millisekkunden	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle
	CMS_TIMEOUT	nach Ablauf der Wartezeit erfolglos

Beschreibung:

Mit e_wait() kann ein Prozeß auf ein Event warten. Er wird suspendiert und nimmt solange nicht am Scheduling teil, bis die Bedingung erfüllt ist, die durch ***opcode*** und den Wert ***value*** spezifiziert wird oder bis die durch den Parameter ***timeout*** spezifizierten Wartezeit abgelaufen ist. Im letzten Fall liefert die Funktion den Returncode CMS_TIMEOUT.

Durch die Werte von ***opcode*** und ***value*** sind die folgenden Bedingungen möglich:

```
      Opcode  Bedingung
      ──────────────────────────────────────────────────────────────────
─────────
      '='  event = value
      '>'  event > value (signed)
      '<'  event < value (signed)
      '&'  event & value (bitweises AND - warte bis bit(s) gesetzt)
      '^'  event ^ value (bitweises XOR).
```

7.2.8. e_getset() - Event setzen

Syntax:

CMS_RETURN e_getset(CMS_HANDLE ehandle, int op, int value, int * oldval)

Parameter:

CMS_HANDLE	ehandle	Event-Handle	IN
int	op	Operation	IN
int	value	Wert	IN
int *	oldval	Feld für den alten Wert	OUT

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Der Wert des Events wird abgefragt und neu gesetzt. Der alte Wert wird über ***oldval*** zurückgegeben (analog zu e_get()). Anschließend wird der Wert mittels ***op*** und ***value*** gesetzt (analog zu e_set()).

Beide Operationen werden dabei zu einer unteilbaren Aktion zusammengefaßt. Das bedeutet, daß zwischen dem Lesen und dem Setzen kein anderer Prozeß dieses Event verändern kann.

7.2.9. e_signal() - Event-Signal definieren

Syntax:

CMS_RETURN e_signal(CMS_HANDLE ehandle, int kind, CMS_SIGNAL signal)

Parameter:

CMS_HANDLE	ehandle	Event-Handle	IN
int	kind	Art des Ereignisses	IN
CMS_SIGNAL	signal	Signalwort	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Bei Eintreten des mit ***kind*** spezifizierten Ereignisses des Events ***ehandle*** wird an den rufendenProzeß das Signal ***signal*** gesendet. Dieses Verhalten bleibt für dieses Event solange bestehen, bis es durch erneuten Aufruf von e_signal() im selben Prozeß wieder geändert wird.

Es ist im Moment nur ein Event-Ereignis definiert, das signalisiert werden kann: das Setzen des Events.

7.2.10. e_status() - Event-Status abfragen

Syntax:

int e_status(CMS_HANDLE ehandle)

Parameter:

CMS_HANDLE	ehandle	Event-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit e_status() kann der aktuelle Zustand eines Events abgefragt werden. Das ist der Inhalt des entsprechenden Integer-Wertes.

7.2.11. e_getcb() - Event Control Block abfragen

Syntax:

CMS_RETURN e_getcb(CMS_HANDLE ehandle, CMS_ECB ** ecb)

Parameter:

CMS_HANDLE	ehandle	Event-Handle	IN
CMS_ECB **	ecb	Feld für Adresse des Kontrollblocks	OUT

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Es wird ein Zeiger auf den Event-Kontrollblock von ***ehandle*** zurückgegeben. Der Kontrollblock darf nicht verändert werden !
Er hat folgenden Aufbau:

```
typedef struct                          /* event control block       */
{
   CMS_SIGNAL siglist[CMS_MAXPROC];     /* processes to be notified  */
   unsigned int opencnt;                /* open counter              */
   int delreq;                          /* delete requested          */
   int val;                             /* event value               */
   PROC wlist[CMS_MAXPROC];             /* proc~s waiting for retry  */
   char name[NAMELEN];                  /* event name                */
   BYTE sigflag;                        /* any process to notify ?   */
} ECB;
```

7.2.12. e_list() - Event-Informationen ausgeben

Syntax:

void e_list(void)

Parameter:

keine

Return:

kein

Beschreibung:

Es wird eine Tabelle mit Event-Informationen ausgegeben. Sie hat folgenden Aufbau:

```
EVENT LIST:
  h name     value    count
  0 ind_evt  $023D        2
  1 3rd_evt  $0239        2
```

Im einzelnen bedeuten die Parameter:

h	Event-Handle
name	Name des Events
value	Wert des Events
count	Anzahl der Benutzer des Events (Open-Zähler)

8. Prozeß Kommunikation

Die CMS-Objekte Queue und Pipe sowie eine bestimmte Klasse von Drivern, die sogenannten Communication Driver, haben ein einheitliches Interface für den Nachrichtentransport. Zwischen allen Objekten, die dieses IPC-Interface (Inter Process Communication) unterstützen, ist die Umlenkung von Nachrichten möglich.

Das IPC-Interface besteht aus den folgenden Funktionen:

..._read()	-	Lesen
..._look()	-	"Ansehen"
..._write()	-	Schreiben
..._flush()	-	Puffer leeren
..._control()	-	Umlenkung definieren
..._signal()	-	Signal definieren

Die _control()-Funktion wird nur unter Queue-Funktionen erläutert; für Pipes und Communication Driver gelten die Erläuterungen entsprechend. Die übrigen Funktionen werden bei den einzelnen Objekten erläutert.

8.1. Queue-Funktionen

8.1.1. q_create() - Queue erzeugen

Syntax:

CMS_HANDLE q_create(char * name, int msgnum, int msgsize)

Parameter:

char *	name	Name der Queue	IN
int	msgnum	Maximale Anzahl der Messages	IN
int	msgsize	Maximale Größe einer Message	IN

Return:

CMS_HANDLE	> 0	Queue-Handle
	CMS_EXIST	Queue existiert schon
	CMS_FULL	Queue-Tabelle ist voll
	CMS_NOMEM	zuwenig Speicher frei

Beschreibung:

Es wird eine Queue ***name*** erzeugt. ***msgsize*** gibt die maximale Messagelänge an, die durch dieses Objekt transportiert werden soll; ***msgnum*** Messages sollten vom System zwischengespeichert werden. Es ist möglich, Messages der Länge ***msgsize*** zu transportieren.
Falls ***msgsize*** = 0 ist, werden nur Pointer auf die Messages transportiert, sonst wird der Pufferinhalt kopiert.
Die Funktion liefert ein Handle (>0) oder einen Errorcode (<0) zurück.

8.1.2. q_delete() - Queue löschen

Syntax:

CMS_RETURN q_delete(CMS_HANDLE qhandle)

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit q_delete() wird eine Queue gelöscht. Dies bedeutet, daß CMS in der Queue-Tabelle den Eintrag für diese Queue löscht. Weitere Zugriffe auf die Queue sind nicht möglich.
Über den Open-Zähler der zu löschenden Queue wird geprüft, ob sie von anderen Prozessen noch benutzt wird (noch geöffnet ist). Ist dies der Fall, wird die Queue nicht gelöscht, sondern zum Löschen vorgemerkt und nach dem letzten Close auf die Queue gelöscht.

8.1.3. q_open() - Queue öffnen

Syntax:

CMS_HANDLE q_open(char * name)

Parameter:

char *	name	Name der Queue	IN

Return:

CMS_HANDLE	qhandle	Queue-Handle
	CMS_NOTFND	Queue nicht gefunden

Beschreibung:

Mit q_open() wird eine Queue geöffnet. Dazu wird der Name der zu öffnenden Queue angegeben. Bei Erfolg meldet CMS das Queue-Handle zurück, welches bei allen nachfolgenden Zugriffen auf die Queue benutzt wird.

Bei jedem Open auf eine Queue wird ihr Open-Zähler inkrementiert; dieser gibt an, wie viele Prozesse die Queue benötigen. Für jeden Aufruf von q_open() in einem Prozeß muß daher auch ein Aufruf von q_close() vor Prozeßende erfolgen, da sonst das Löschen der Queue mit q_delete() fehlschlägt.

8.1.4. q_close() - Queue schließen

Syntax:

CMS_RETURN q_close(CMS_HANDLE qhandle)

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit q_close() wird der Zugriff auf eine Queue beendet. Ein weiterer Zugriff ist erst nach erneutem Open möglich.

Bei jedem Close wird der Open-Zähler für diese Queue dekrementiert. Dieser gibt an, ob eine Queue noch von einem Prozeß benötigt wird.

8.1.5. q_handle() - Nächstes Queue-Handle liefern

Syntax:

CMS_HANDLE q_handle(CMS_HANDLE qhandle)

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN

Return:

CMS_ HANDLE	> 0	nächstes Queue-Handle
	= 0	kein weiteres Handle vorhanden

Beschreibung:

Diese Funktion liefert zu einem gültigen Queue-Handle das nächste gültige Queue-Handle. Für ***qhandle*** = 0 wird das erste Queue-Handle geliefert, für das letzte gültige Queue-Handle wird 0 zurückgeliefert.

8.1.6. q_read() - Aus Queue lesen

Syntax:

int q_read(CMS_HANDLE qhandle, char * buf, int buflen, CMS_TIME timeout)

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN
char *	buf	Message-Puffer	OUT
int	buflen	maximale Pufferlänge	IN
CMS_TIME	timeout	Wartezeit in Millisekunden	IN

Return:

int	>= 0	Anzahl der gelesenenen Bytes
	CMS_BADHANDLE	Ungültiges Handle
	CMS_SIZE	Buffer für Message zu klein
	CMS_TIMEOUT	Wartezeit abgelaufen, kein Erfolg
	CMS_EMPTY	keine Message vorhanden

Beschreibung:

Mit q_read() wird genau eine Message aus der angegebenen Queue gelesen. Dazu werden ein Message-Buffer ***buf***, in dem die gelesene Message abgelegt wird, und dessen Größe ***buflen*** spezifiziert.

Befindet sich zum Zeitpunkt des Aufrufs keine Message in der Queue, wird der Wert des Parameters timeout überprüft. Ist er größer Null, wird der rufende Prozeß suspendiert, bis eine Message empfangen wird oder die durch ***timeout*** angegebene Zeit abläuft. Im letzten Fall liefert die Funktion den Returncode CMS_TIMEOUT zurück. Ist der Wert von ***timeout*** gleich Null- kehrt die Funktion sofort mit dem Returncode CMS_EMPTY zurück.
Ist der angegebene Buffer zu klein, wird nicht gelesen. Die Funktion kehrt dann mit dem Returncode CMS_SIZE zurück.
Bei erfolgreichem Lesen wird als Returnvalue die Länge der gelesenen Message zurückgemeldet.

8.1.7. q_look() - Daten aus Queue "ansehen"

Syntax:

int q_look(CMS_HANDLE qhandle, char * buf, int buflen)

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN
char *	buf	Message-Puffer	OUT
int	buflen	maximale Pufferlänge	IN

Return:

int	>= 0	Länge der Message in der Queue
	CMS_BADHANDLE	Ungültiges Handle
	CMS_EMPTY	keine Message vorhanden
	CMS_ERROR	Länge negativ

Beschreibung:

Mit q_look() wird die nächste Message oder ein Teil davon gelesen, ohne daß diese aus der Queue entfernt wird. Ist keine Message vorhanden, liefert die Funktion den Returncode CMS_EMPTY zurück.

Ist der angegebene Buffer kleiner als die gesamte Message, werden die ersten ***buflen*** Bytes gelesen. ***buflen*** darf auch 0 sein.

Ist eine Message vorhanden, liefert q_look() deren Länge zurück. Sie kann verwendet werden, um den Message-Buffer des Prozesses ausreichend zu dimensionieren.

8.1.8. q_write() - In Queue schreiben

Syntax:

int q_write(CMS_HANDLE qhandle, char * buf, int msglen, CMS_TIME timeout)

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN
char *	buf	Message-Puffer	IN
int	msglen	Message-Länge	IN
CMS_TIME	timeout	Wartezeit in Millisekunden	IN

Return:

int	>= 0	Anzahl der geschriebenen Bytes
	CMS_BADHANDLE	Ungültiges Handle
	CMS_SIZE	Message für Zielpuffer zu groß
	CMS_TIMEOUT	Wartezeit abgelaufen, kein Erfolg

Beschreibung:

Mit q_write() wird eine Message in eine Queue geschrieben. Dazu werden der Message-Buffer ***buf*** und die Länge ***msglen*** der darin gespeicherten Message übergeben.

Ist zum Zeitpunkt des Aufrufs in der spezifizierten Queue ***qhandle*** nicht genügend Platz für die Aufnahme der Message, wird der Parameter ***timeout*** ausgewertet. Ist der Wert ungleich Null, wird der rufende Prozeß suspendiert, bis die Message geschrieben werden kann oder die durch ***timeout*** angegebene Zeit abgelaufen ist. Im letzten Fall liefert die Funktion den Returncode CMS_TIMEOUT zurück. Ist der Wert von ***timeout*** gleich Null, kehrt die Funktion sofort mit dem Returncode 0 zurück.

Bei erfolgreichem Schreiben wird als Returnvalue die Länge der geschriebenen Message zurückgemeldet.

8.1.9. q_writeprio() - Priorisiert in Queue schreiben

Syntax:

```
int q_writeprio( CMS_HANDLE qhandle, char * buf, int buflen, CMS_TIME timeout,
                                                    int prio )
```

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN
char *	buf	Message-Puffer	IN
int	buflen	maximale Pufferlänge	IN
CMS_TIME	timeout	Wartezeit in Millisekunden	IN
int	prio	Priorität der Message	IN

Return:

int	>= 0	Anzahl der geschriebenen Bytes
	CMS_BADHANDLE	Ungültiges Handle
	CMS_SIZE	Message für Zielpuffer zu groß
	CMS_TIMEOUT	Wartezeit abgelaufen, kein Erfolg

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Es wird wie mit q_write() eine Nachricht in die Queue geschrieben, diese wird aber nicht unbedingt an das Ende der Queue eingereiht. Vielmehr wird diese Nachricht entsprechend ihrer "Priorität" ***prio*** eingefügt: Die Nachricht kommt hinter alle Nachrichten mit größerer oder gleich großer Priorität und vor alle Nachrichten mit kleinerer Priorität.

Die kleinste mögliche Priorität ist 0. Nachrichten, die mit q_write() geschrieben werden, haben die Priorität 0. Diese Nachrichten-Priorität ist nicht mit der Prozeß-Priorität zu verwechseln.

8.1.10. q_flush() - Queue leeren

Syntax:

CMS_RETURN q_flush(CMS_HANDLE qhandle)

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit q_flush() kann eine Queue geleert werden. Dadurch werden alle in der Queue befindlichen Messages gelöscht, ihr Inhalt geht verloren.

8.1.11. q_control() - Queue-Umlenkung definieren

Syntax:

```
CMS_HANDLE q_control( CMS_HANDLE qhandle, CMS_HANDLE chandle,
                     CMS_CONTROL * icontrol, CMS_HANDLE * dest)
```

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN
CMS_HANDLE	chandle	Control-Handle	IN
CMS_CONTROL *	icontrol	Datenstruktur mit den Umlenkungsparametern	IN
CMS_HANDLE *	dest	Liste von Zielinstanzen	IN

Return:

CMS_HANDLE	> 0	neues Control -Handle
	CMS_FULL	zu viele Redirection-Anweisungen
	CMS_NOMEM	kein Speicher mehr frei
	CMS_ERROR	anderer Fehler (zu viele Filterbedingungen, Ziele o.ä.)

Beschreibung:

Die Nachrichten in der Inter Process Communication können umgelenkt werden. Dabei ist es möglich, Nachrichten auch in ein CMS-"Objekt" eines anderen Typs (z.B. Communication Driver, s.u.) umzulenken. Die Umlenkung geschieht transparant für die an der Kommunikation beteiligten Prozesse.

qhandle gibt die zu beeinflussende Queue an. Für jede Queue können mehrere Umlenkungsbedingungen formuliert werden, diese werden mittels eines sogenannten Control-Handles unterschieden. q_control() liefert das entsprechende Control-Handle zurück. Mit ***chandle*** = 0 wird eine neue Umlenkungbedingung definiert, wird ***chandle*** gleich einem bekannten Control-Handle gesetzt, dann wird diese Umlenkungsbedingung durch den Aufruf geändert (z.B. gelöscht).

Ein Datenstrom kann an null, ein oder mehrere Ziele umgelenkt werden. Sollen die Daten auch an die ursprüngliche Zielinstanz gehen, so ist diese als Ziel mit anzugeben. Der Parameter ***dest*** ist die Liste der Ziele, die durch ihre Handles angegeben werden; Die Liste wird durch das 'Handle' 0 abgeschlossen; statt einer leeren Liste ist auch ein NULL-Pointer zulässig.

Die Umlenkung kann in Abhängigkeit vom Nachrichteninhalt erfolgen; dafür können Filterbedingungen angegeben werden. Nachrichten, die keine Filterbedingung dieser Queue erfüllen, werden normal zugestellt. Über den Parameter ***icontrol*** wird die Filterbedingung eingestellt. Ein NULL-Pointer entspricht hier der Bedingung (unconditional) FALSE, das heißt die Bedingung wird durch keine Nachricht erfüllt.

Mit dem Aufruf

q_control(qhandle, chandle, NULL, NULL);

kann die Umlenkungsbedingung ***chandle*** gelöscht werden.

icontrol ist eine Liste von Einträgen des Typs CMS_CONTROL. Jeder Eintrag ist ein Teil der Filterbedingung. Die Filterbedingung wird als logische Verknüpfung von Vergleichen gebildet; jeder Eintrag in der ***icontrol***-Liste entspricht einem Vergleich. Dabei wird ein Ausschnitt aus der Nachricht durch seine Position in der Nachricht und den Typ (1, 2 oder 4 Byte; signed oder unsigned) definiert und mit einem festen Wert verglichen. Die logische Verknüpfung der Vergleiche wird in 'short-circuit'-Form notiert, d.h. statt der AND- bzw. OR-Operatoren werden bedingte Sprung-Operatoren angegeben.

Der Typ CMS_CONTROL ist folgendermaßen definiert:

```
typedef struct
{
   /* position in der Nachricht: [ind_offset] + offset */
   BYTE ind_type;            /* 0 oder Typ des Indirekt-Pointers         */
   unsigned int ind_offset;  /* Absolute Position des Indirekt-Pointers  */
   unsigned int offset;      /* Offset                                   */
   BYTE   type;              /* Typ des Ausschnittes FILT_BYTE, ...      */
   ULONG  value;             /* Vergleichswert                           */
   BYTE   jump;              /* Sprung: FILT_JEQ, FILT_JNE, ...          */
   SHORT  jdest;             /* Sprungziel: Index in der icontrol-Liste  */
} CMS_CONTROL
```

Ein Eintrag in der Liste ***icontrol*** wird abgearbeitet, indem der aktuelle Wert (s.u.) verglichen wird mit dem eingespeicherten Vergleichswert. Wenn die Sprungbedingung erfüllt ist, wird innerhalb dieser Liste bei dem Index weitergearbeitet, der in ***jdest*** angegeben ist, ansonsten wird der nächste Eintrag abgearbeitet. Der Index beginnt bei 0.

Die Liste muss mit FILT_END beendet werden.

FILT_TRUE und FILT_FALSE bezeichnen den Erfolg oder Mißerfolg der Vergleichsserie. Die Sprungbedingungen sind:

FILT_JEQ	Sprung, wenn aktueller Wert	= Vergleichswert
FILT_JNE	Sprung, wenn	!=
FILT_JLT	Sprung, wenn	<
FILT_JLE	Sprung, wenn	<=
FILT_JGT	Sprung, wenn	>
FILT_JGE	Sprung, wenn	>=

Typangaben:

FILT_BYTE	1 Byte unsigned
FILT_WORD	2 Bytes unsigned
FILT_DWORD	4 Bytes unsigned
FILT_BYTE | FILT_SIGNED	1 Byte signed
FILT_WORD | FILT_SIGNED	2 Bytes signed

FILT_DWORD | FILT_SIGNED 4 Bytes signed

Die Postion des aktuellen Wertes relativ zum Anfang der Nachricht kann absolut angegeben werden:
ind_type = 0
Position = *offset*

Die Position kann aber auch indirekt angegeben werden. Bei der indirekten Angabe wird die Position aus einer Zahl, die in der Nachricht steht, bestimmt. *ind_offset* ist dann die Position dieser Zahl in der Nachricht, *ind_type* ist deren Typ. Die Position ergibt sich zu:
Position = **ind_offset* + *offset*

Beispiel:

Am Anfang der Nachricht stehe ein Feld filter_info mit der Struktur:

```
typedef struct
{
      BYTE ps;                /* pos = 0  lng = 1 */
      BYTE reserve;           /* pos = 1  lng = 1 */
      SHORT pt;               /* pos = 2  lng = 2 */
      LONG snd;               /* pos = 4  lng = 4 */
      LONG rcv;               /* pos = 8  lng = 4 */
} FILTER_INFO;
```

Es sollen nun Nachrichten umgelenkt werden, für die gilt:

ps=200 und
22 <= pt <= 33 und
(snd=1234 oder snd=4321) und
(rcv=1234 oder rcv=4321)

Als logischer Operator, wobei (buf+lng) die Position des aktuellen Wertes beschreibt, ergibt das:

```
(*(UCHAR *)(buf + 0) == 200 )                                              &&
(*(USHORT *)(buf + 2) >= 22)                                               &&
(*(USHORT *)(buf + 2) <= 33)                                               &&
(  (*(ULONG *)(buf + 4) == 1234)     ||    (*(ULONG *)(buf + 4) == 4321) )    &&
(  (*(ULONG *)(buf + 8) == 1234)     ||    (*(ULONG *)(buf + 8) == 4321) )
```

Es sollen also nur die Nachrichten umgelenkt werden, für die diese Bedingung erfüllt ist; alle anderen Nachrichten sollen normal zugestellt werden(wenn für sie auch keine andere Filterbedingung zutrifft).

Diese Darstellung läßt sich unmittelbar in eine Tabelle überführen, wie sie von q_control() gefordert wird. Aus Platzgründen sind hier die in CMS.H definierten Konstanten FILT_TRUE, FILT_FALSE, FILT_JNE usw. durch TRUE, FALSE, JNE usw. ersetzt.

Index	**ind_type**	**offset**	**type**	**value**	**jump**	**jdest**	**Pseudocode** (Nachricht: BYTE buf[])

0	0	0	BYTE	200	JNE	8	if(*(UCHAR *)(buf + 0) != 200) goto 8
1	0	2	WORD	22	JLT	8	if(*(USHORT *)(buf + 2) < 22) goto 8
2	0	2	WORD	33	JGT	8	if(*(USHORT *)(buf + 2) > 33) goto 8
3	0	4	DWORD	1234	JEQ	5	if(*(ULONG *)(buf + 4) == 1234) goto 5
4	0	4	DWORD	4321	JNE	8	if(*(ULONG *)(buf + 4) != 4321) goto 8
5	0	8	DWORD	1234	JEQ	7	if(*(ULONG *)(buf + 8) == 1234) goto 7
6	0	8	DWORD	4321	JNE	8	if(*(ULONG *)(buf + 8) != 4321) goto 8
7					TRUE		/* Filterbedingung erfüllt -> Umlenken */
8					FALSE		/* Filterbedingung nicht erfüllt */
9					END		/* Ende der Liste */

8.1.12. q_signal() - Queue-Signal definieren

Syntax:

int q_signal(CMS_HANDLE qhandle, int kind, CMS_SIGNAL signal)

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN
int	kind	Art des Ereignisses	IN
CMS_SIGNAL	signal	Signalwort	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Bei Eintreten des mit ***kind*** spezifizierten Ereignisses der Queue ***qhandle*** wird an den rufenden Prozeß das Signal ***signal*** gesendet. Dieses Verhalten bleibt für diese Queue solange bestehen, bis es durch erneuten Aufruf von q_signal() im selben Prozeß wieder geändert wird.

Folgende Queue-Ereignisse können signalisiert werden:

kind = CMS_CANREAD : Aus der Queue kann gelesen werden.
kind = CMS_CANWRITE : In die Queue kann geschrieben werden.

Jeder Prozess kann nur ein Signal pro Queue definieren. Die beiden Arten können aber (mit |) kombiniert werden.

8.1.13. q_status() - Queue-Status abfragen

Syntax:

int q_status(CMS_HANDLE qhandle)

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN

Return:

int	>= 0	Anzahl der Messages in der Queue
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit q_status() kann der aktuelle Zustand (Füllstand) der Queue **qhandle** abgefragt werden. Es wird die Anzahl der Messages in der Queue zurückgemeldet.

8.1.14. q_getcb() - Queue Control Block abfragen

Syntax:

CMS_RETURN q_getcb(CMS_HANDLE qhandle, CMS_QCB ** qcb)

Parameter:

CMS_HANDLE	qhandle	Queue-Handle	IN
CMS_QCB **	qcb	Feld für Queue-Controlblock	OUT

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Es wird ein Zeiger auf den Queue-Kontrollblock von ***qhandle*** zurückgegeben. Der Kontrollblock darf nicht verändert werden !
Er hat folgenden Aufbau:

```
typedef struct                          /* queue control block       */
{
   MTAB * mtp;                          /* pointer to message table  */
   char * buf;                          /* queue buffer              */
   CMS_REDIR * red;                     /* addr of redirection list  */
   int msgsize;                         /* max message length        */
   int msgnum;                          /* max number of messages    */
   int size;                            /* queue buffer size and
                                           mode indicator            */
   unsigned int opencnt;                /* open counter              */
   int delreq;                          /* delete requested          */
   SIGLIST siglist[CMS_MAXPROC];        /* list of signals           */
   PROC rlist[CMS_MAXPROC];             /* waiting for read proc.s   */
   PROC wlist[CMS_MAXPROC];             /* waiting for write proc.s  */
   char name[NAMELEN];                  /* queue name                */
} QCB;
```

8.1.15. q_list() - Queue-Informationen ausgeben

Syntax:

void q_list(void)

Parameter:

keine

Return:

kein

Beschreibung:

Es werden Informationen über alle Queues in folgender Tabellenform ausgegeben:

```
QUEUE LIST:
  h name      buffer       msgtab       size  status
  0 queue1   $22D5C72E    $22D5CBE6     1200       1
  1 queue2   $22D5C20E    $22D5C6C6     1200       0
```

Im einzelnen bedeuten die Parameter:

h	Queue-Handle
name	Name der Queue
buffer	Adresse der ersten Message (wenn Queue im Kopiermodus, sonst = 0)
msgtab	Adresse der Message-Liste (Zeiger auf die Messages)
size	Größe der Queue (wenn Queue im Kopiermodus, sonst = 0)
status	Anzahl der Messages in der Queue

8.2. Pipe-Funktionen

8.2.1. m_create() - Pipe erzeugen

Syntax:

CMS_HANDLE m_create(char * name, int size)

Parameter:

char *	name	Name der Pipe	IN
int	size	Größe der Pipe in Bytes	IN

Return:

CMS_HANDLE	mhandle	Pipe-Handle
	CMS_EXIST	Pipe existiert schon
	CMS_FULL	Pipe-Tabelle voll
	CMS_NOMEM	zuwenig Speicher

Beschreibung:

Es wird ein Pipe ***name*** erzeugt. ***size*** gibt die Puffergröße an.
Die Funktion liefert ein Handle (>0) oder einen Errorcode (<0) zurück.
Es können CMS_MAXPIPES Pipes erzeugt werden (Konstante in CMS.H)

8.2.2. m_delete() - Pipe löschen

Syntax:

CMS_RETURN m_delete(CMS_HANDLE mhandle)

Parameter:

CMS_HANDLE	mhandle	Pipe-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit m_delete() wird eine Pipe gelöscht. Dies bedeutet, daß CMS in der Pipe-Tabelle den Eintrag für diese Pipe löscht. Weitere Zugriffe auf die Pipe sind nicht möglich.
Über den Open-Zähler der zu löschenden Pipe wird geprüft, ob sie von anderen Prozessen noch benutzt wird (noch geöffnet ist). Ist dies der Fall, wird die Pipe nicht gelöscht, sondern zum Löschen vorgemerkt und nach dem letzten Close auf die Pipe gelöscht.

8.2.3. m_open() - Pipe öffnen

Syntax:

CMS_HANDLE m_open(char * name)

Parameter:

char *	name	Name der Pipe	IN

Return:

CMS_HANDLE	mhandle	Pipe-Handle
	CMS_NOTFND	Pipe nicht gefunden

Beschreibung:

Mit m_open() wird eine Pipe geöffnet. Dazu wird der Name der zu öffnenden Pipe angegeben. Bei Erfolg meldet CMS das Pipe-Handle zurück, welches bei allen nachfolgenden Zugriffen auf die Pipe benutzt wird.

Bei jedem Open auf eine Pipe wird ihr Open-Zähler inkrementiert; dieser gibt an, wie viele Prozesse die Pipe benötigen. Für jeden Aufruf von m_open() in einem Prozeß muß daher auch ein Aufruf von m_close() vor Prozeßende erfolgen, da sonst das Löschen der Pipe mit m_delete() fehlschlägt.

8.2.4. m_close() - Pipe schließen

Syntax:

CMS_RETURN m_close(CMS_HANDLE mhandle)

Parameter:

CMS_HANDLE	mhandle	Pipe-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit m_close() wird der Zugriff auf eine Pipe beendet. Ein weiterer Zugriff ist erst nach erneutem Open möglich.

Bei jedem Close wird der Open-Zähler für diese Pipe dekrementiert. Dieser gibt an, ob eine Pipe noch von einem Prozeß benötigt wird.

8.2.5. m_handle() - Nächstes Pipe-Handle liefern

Syntax:

CMS_HANDLE m_handle(CMS_HANDLE mhandle)

Parameter:

CMS_HANDLE	mhandle	Pipe-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Diese Funktion liefert zu einem gültigen Pipe-Handle das nächste gültige Pipe-Handle. Für ***mhandle*** = 0 wird das erste Pipe-Handle geliefert, für das letzte gültige Pipe-Handle wird 0 zurückgeliefert.

8.2.6. m_read() - Aus Pipe lesen

Syntax:

int m_read(CMS_HANDLE mhandle, char * buf, int buflen, CMS_TIME timeout)

Parameter:

CMS_HANDLE	mhandle	Pipe-Handle	IN
char *	buf	Daten-Puffer	OUT
int	buflen	maximale Pufferlänge	IN
CMS_TIME	timeout	Wartezeit in Millisekunden	IN

Return:

int	>= 0	Anzahl der gelesenenen Bytes
	CMS_BADHANDLE	Ungültiges Handle
	CMS_TIMEOUT	Wartezeit abgelaufen, kein Erfolg
	CMS_EMPTY	keine Message vorhanden

Beschreibung:

Mit m_read() wird eine vorgegebene Byteanzahl aus der Pipe ***mhandle*** gelesen. Dazu werden ein Buffer ***buf***, in dem die gelesenen Bytes abgelegt werden, und seine Größe ***buflen*** spezifiziert.

Befinden sich zum Zeitpunkt des Aufrufs keine Daten in der Pipe, wird der Wert des Parameters ***timeout*** überprüft. Ist er größer Null, wird der rufende Prozeß suspendiert, bis Daten empfangen werden oder die durch ***timeout*** angegebene Zeit abläuft. Im letzten Fall liefert die Funktion den Returncode CMS_TIMEOUT zurück. Ist der Wert von ***timeout*** gleich Null- kehrt die Funktion sofort mit dem Returncode CMS_EMPTY zurück.
Bei erfolgreichem Lesen wird als Returnvalue die Länge der gelesenen Message zurückgemeldet.

8.2.7. m_look() - Daten aus Pipe "ansehen"

Syntax:

int m_look(CMS_HANDLE mhandle, char * buf, int buflen)

Parameter:

CMS_HANDLE	mhandle	Pipe-Handle	IN
char *	buf	Daten-Puffer	OUT
int	buflen	maximale Pufferlänge	IN

Return:

int	>= 0	Anzahl der Bytes in der Pipe
	CMS_BADHANDLE	Ungültiges Handle
	CMS_EMPTY	keine Message vorhanden

Beschreibung:

m_look() liest eine vorgegebene Byteanzahl aus der Pipe ***mhandle*** , sofern entsprechend viel Daten vorhanden sind. Im Unterschied zu m_read() wird nicht die Anzahl der gelesenen Bytes, sondern die Anzahl der in der Pipe vorhandenen Bytes zurückgeliefert, und die Daten bleiben im internen Buffer erhalten. Das bedeutet, ein nachfolgendes m_read() liefert dieselben Daten

8.2.8. m_write() - In Pipe schreiben

Syntax:

int m_write(CMS_HANDLE mhandle, char * buf, int buflen, CMS_TIME timeout)

Parameter:

CMS_HANDLE	mhandle	Pipe-Handle	IN
char *	buf	Message-Puffer	IN
int	buflen	maximale Pufferlänge	IN
CMS_TIME	timeout	Wartezeit in Millisekunden	IN

Return:

int	>= 0	Anzahl der geschriebenen Bytes
	CMS_BADHANDLE	Ungültiges Handle
	CMS_SIZE	nicht genügend freier Platz in der Pipe
	CMS_TIMEOUT	Wartezeit abgelaufen, kein Erfolg

Beschreibung:

Mit m_write() wird ein Datenfeld in eine Pipe geschrieben. Dazu werden der Daten-Buffer ***buf*** und die Länge ***buflen*** übergeben.

Ist zum Zeitpunkt des Aufrufs in der spezifizierten Pipe ***mhandle*** nicht genügend Platz für die Aufnahme der Daten , wird der Parameter ***timeout*** ausgewertet. Ist der Wert ungleich Null, wird der rufende Prozeß suspendiert, bis die Message geschrieben werden kann oder die durch ***timeout*** angegebene Zeit abgelaufen ist. Im letzten Fall liefert die Funktion den Returncode CMS_TIMEOUT zurück. Ist der Wert von ***timeout*** gleich Null, kehrt die Funktion sofort mit dem Returncode CMS_SIZE zurück. Es wird also entweder alles oder nichts geschrieben.

Bei erfolgreichem Schreiben wird als Returnvalue die Anzahl der geschriebenen Bytes zurückgeliefert.

8.2.9. m_flush() - Pipe leeren

Syntax:

CMS_RETURN m_flush(CMS_HANDLE mhandle)

Parameter:

CMS_HANDLE	mhandle	Pipe-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit m_flush() kann eine Pipe geleert werden. Der Inhalt der Pipe geht verloren.

8.2.10. m_control() - Pipe-Umlenkung definieren

Syntax:

CMS_HANDLE m_control(CMS_HANDLE mhandle, CMS_HANDLE chandle,
CMS_CONTROL * icontrol, CMS_HANDLE * dest)

Parameter:

CMS_HANDLE	mhandle	Pipe-Handle	IN
CMS_HANDLE	chandle	Control-Handle	IN
CMS_CONTROL *	icontrol	Datenstruktur mit den Umlenkungsparametern	IN
CMS_HANDLE *	dest	Liste von Zielinstanzen	IN

Return:

CMS_HANDLE	> 0	neues Control -Handle
	CMS_FULL	zu viele Redirection-Anweisungen
	CMS_NOMEM	kein Speicher mehr frei
	CMS_ERROR	anderer Fehler (zu viele Filterbedingungen, Ziele o.ä.)

Beschreibung:

Mit der Funktion m_control() werden die IPC-Umleitungen für die Pipe ***mhandle*** definiert. Damit wird CMS veranlaßt, nachfolgende Daten, die ein Prozeß oder Driver über m_write() an die Pipe ***mhandle*** sendet, selektiv und vom Aufrufer unbemerkt umzulenken. Als neue Ziele können Queues, Pipes oder Driver angegeben werden.

Die anderen Parameter entsprechen denen von q_control() und sind dort ausführlich erläutert, ebenso die Wirkung der Control-Struktur.

8.2.11. m_signal() - Pipe-Signal definieren

Syntax:

int m_signal(CMS_HANDLE mhandle, int kind, CMS_SIGNAL signal)

Parameter:

CMS_HANDLE	mhandle	Pipe-Handle	IN
int	kind	Art des Ereignisses	IN
CMS_SIGNAL	signal	Signalwort	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Bei Eintreten des mit ***kind*** spezifizierten Ereignisses der Pipe ***mhandle*** wird an den rufenden Prozeß das Signal ***signal*** gesendet. Dieses Verhalten bleibt für diese Pipe solange bestehen, bis es durch erneuten Aufruf von m_signal() im selben Prozeß wieder geändert wird.

Folgende Pipe-Ereignisse können signalisiert werden:

kind = CMS_CANREAD : Aus der Pipe kann gelesen werden.
kind = CMS_CANWRITE : In die Pipe kann geschrieben werden (wenigstens 1 Byte).

Jeder Prozess kann nur ein Signal pro Pipe definieren. Die beiden Arten können aber (mit |) kombiniert werden.

8.2.12. m_status() - Pipe-Status abfragen

Syntax:

int m_status(CMS_HANDLE mhandle)

Parameter:

CMS_HANDLE	mhandle	Pipe-Handle	IN

Return:

int	>= 0	Anzahl der Bytes in der Pipe
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Mit m_status() kann der aktuelle Zustand (Füllstand) der Pipe **mhandle** abgefragt werden. Es wird die Anzahl der Bytes in der Pipe zurückgemeldet.

8.2.13. m_getcb() - Pipe Control Block abfragen

Syntax:

int m_getcb(CMS_HANDLE mhandle, CMS_MCB ** mcb)

Parameter:

CMS_HANDLE	mhandle	Pipe-Handle	IN
CMS_MCB **	mcb	Feld für Pipe-Controlblock	OUT

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Es wird ein Zeiger auf den Pipe-Kontrollblock von ***mhandle*** zurückgegeben. Der Kontrollblock darf nicht verändert werden !
Er hat folgenden Aufbau:

```
typedef struct                          /* pipe control block              */
{
   CMS_REDIR * red;                     /* redirection orders        */
   BYTE *buf;                           /* pipe buffer               */
   BYTE *bufend;                        /* pipe buffer end           */
   unsigned int opencnt;                /* open counter              */
   unsigned int in;                     /* in pointer                */
   unsigned int out;                    /* out pointer               */
   int size;                            /* pipe buffer size          */
   int delreq;                          /* delete requested          */
   PROC rlist[CMS_MAXPROC];             /* waiting for read proc's   */
   PROC wlist[CMS_MAXPROC];             /* waiting for write proc's  */
   SIGLIST siglist[CMS_MAXPROC];        /* proc waiting for signals  */
   char name[NAMELEN];                  /* pipe name                 */
} MCB;
```

8.2.14. m_list() - Pipe-Informationen ausgeben

Syntax:

void m_list(void)

Parameter:

keine

Return:

kein

Beschreibung:

Es wird eine Tabelle mit Informationen zu allen Pipes ausgegeben. Sie hat die Form:

```
PIPE LIST:
    h name     bufadr     bufsize      in      out  status
24576 pipe2    $22D5C846     1024       0       0       0
```

Im einzelnen bedeuten die Parameter:

h	Pipe-Handle
name	Name der Pipe
bufadr	Adresse des Datenpuffers
bufsize	Größe des Datenpuffers
in	Zeiger auf den Anfang des belegten Datenbereichs
out	Zeiger auf das Ende des belegten Datenbereichs (kann kleiner sein als `in`, da die Pipe als Ringbuffer organisiert ist)
status	Anzahl der Bytes in der Pipe

9. Driver

Für einen Driver unter CMS sind zwei Gruppen von Funktionen obligatorisch:

- Driver-Management-Funktionen sind die Funktionen innerhalb von CMS, die zur Verwaltung der Driver-Objekte (Initialisieren, Öffnen, Schließen, Löschen , Listen) benötigt werden oder zum Aufruf der Nutz-Funktionen, die spezifisch für den jeweiligen Driver sind, verwendet werden. Die Funktionsnamen sind d_...(). Die Driver-Management-Funktionen rufen ihrerseits entsprechende Funktionen des konkreten Drivers auf.

  Für die Communication Driver (siehe CMS-Objekte/Driver und unten) steht außerdem das IPC-Interface (Inter-Prozeß-Kommunikation) zur Verfügung. Dieses macht den Aufruf der entsprechenden Standard-Driver-Funktionen komfortabler; u.a. wird die Timeout-Behandlung von CMS übernommen.

- Standard-Driver-Funktionen sind Funktionen, die ein Driver zur Verfügung stellen muß, weil sie von den Driver-Management-Funktionen vorausgesetzt und aufgerufen werden. Dabei wird weiter unterschieden zwischen Standard-Funktionen für alle Driver und Funktionen, die die Communication Driver betreffen.

  Für alle Driver müssen die Funktionen vorhanden sein, die beim Initialisieren, Öffnen, Schließen etc. eines Drivers von den Driver-Management-Funktionen aufgerufen werden. Die Communication Driver müssen zusätzlich Schreib- und Lese- und Signal-Funktionen enthalten.

  Die Standard-Driver-Funktionen werden im folgenden mit den Funktionsnamen drv_...() angegeben, der wirkliche Name hängt vom Driver ab. (Z.B. heißt die hier angegebene Funktion drv_write() im Driver EMI emi_write().)

  Die Driver-Funktionen werden immer über drv_call(Funktionsnummer, ...) aufgerufen. Die Standard-Driver-Funktionen entsprechen also Standard-Funktionsnummern (DRV_INIT, ...).

Außer den genannten Funktionen kann jeder Driver noch weitere, für ihn spezifische Funktionen exportieren. Sie werden nach demselbem Schema wie die Standard-Driver-Funktionen aufgerufen (s.a. Kap. “Programme unter CMS”)

9.1. Driver-Management-Funktionen

9.1.1. d_create() - Driver anlegen

Syntax:

```
CMS_HANDLE d_create( char * name, CMS_DRVENTRY drv_call, int flags, int devs,
                    void * data, unsigned int datalen, int argc, char ** argv )
```

Parameter:

char *	name	Driver-Name	IN
CMS_DRVENTRY	drv_call	Adresse der Einsprung-Funktion	IN
int	flags	Konfigurationsflags	IN
int	devs	Anzahl Devices	IN
void *	data	Datensegment-Adresse	IN
unsigned int	datalen	Länge des Datenbereichs	IN
int	argc	Anzahl der Kommandozeilenargumente	IN
char **	argv	Zeiger auf Liste der Kommandozeilenargumente	IN

Return:

CMS_HANDLE	> 0	Driver-Handle
	CMS_EXIST	ein Driver mit diesem Namen existiert schon
	CMS_FULL	Driver-Tabelle ist voll
	CMS_NOMEM	kein Speicher mehr frei (z.B. für eigenes Datensegment)
	CMS_ERROR	anderer Fehler (z.B. unsinniger devs-Wert)
	andere < 0	Returncode von xyz_init()

Beschreibung:

d_create() wird vom Driver-Code aus aufgerufen. Der Driver ***name*** wird angelegt. CMS werden die Entry-Funktion des Drivers (***drv_call***) und die Anzahl der unterstützten Devices (***devs***) mitgeteilt. Wenn eine Unterstüzung mehrerer Devices nicht benötigt wird, dann sollte ***devs*** = 0 gesetzt werden. (***devs*** wird nur beachtet, wenn in ***flags*** CMSDRV_DEVICES gesetzt ist !)

Über ***flags*** werden die Eigenschaften des Drivers festgelegt. ***flags*** kann aus den Werten CMSDRV_NONSHARED, CMSDRV_DEVICES, CMSDRV_IPC, CMSDRV_MONITOR kombiniert werden. Die Bedeutung dieser Werte und der damit verbundenen Datenmodelle ist im Kapitel “3.2.4. Driver-Konfigurationen” erklärt.

Über den Parameter ***data*** teilt der Driver CMS die Anfangsadresse seines Datensegmentes mit. Dieses Datensegment wird im Shared Data Model für alle Aufrufe verwendet; im Nonshared Data Model dient dieses initiale Datensegment des Drivers nur als "Muster": Für jeden Prozeß, der den Driver aufruft, wird von

CMS ein eigenes Datensegment alloziert und mit den Daten aus dem initialen Datensegment gefüllt, bevor der Driver das erste Mal aufgerufen wird (mit drv_open()).

datalen ist die Länge des initialen Datensegmentes (in Bytes), sie wird nur im Nonshared Data Model benötigt.

Das zurückgelieferte Driver-Handle kann verwendet werden, um den Driver wieder zu löschen (d_delete()). Falls ***devs*** > 0 angegeben wurde, muß zum Lesen oder Schreiben jedes benutzte Device mit d_open() geöffnet werden.

Die Funktion d_create() wird im mitgelieferten Module drvmain.c aufgerufen. Der Entwickler eines Drivers braucht sich darum nicht zu kümmern.

CMS ruft die Driver-Funktion drv_init() mit den Parametern argc und argv auf (über drv_call(DRV_INIT)).

9.1.2. d_delete() - Driver löschen

Syntax:

CMS_RETURN d_delete(CMS_HANDLE dhandle)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Der Driver ***dhandle*** wird gelöscht.

CMS ruft die Driver-Funktion drv_exit() auf (über drv_call(DRV_EXIT)), so daß anwendungsspezifische Aufräumarbeiten vorgenommen werden können. Danach werden alle eventuell noch allozierten Datensegmente des Drivers wieder freigegeben.

9.1.3. d_open() - Driver öffnen

Syntax:

CMS_HANDLE d_open(char * name, int dev)

Parameter:

char *	name	Name des Drivers	IN
int	dev	Gerätenummer	IN

Return:

CMS_HANDLE	> 0	Driver-Handle
	CMS_NOTFND	Driver nicht gefunden
	CMS_NOMEM	Nicht genügend Speicher
	CMS_ERROR	ungültige Devce-Nummer
	andere < 0	Returncode von xyz_open()

Beschreibung:

Der Driver ***name*** wird geöffnet, bei Erfolg wird das Handle zurückgegeben. ***dev*** ist die Nummer des Devices. Bei Drivern mit mehreren Devices wird für jedes Device ein eigenes Handle zurückgeliefert.

Falls für den Driver eines der Flags CMSDRV_NONSHARED oder CMSDRV_DEVICES gesetzt ist, ruft CMS die Driver-Funktion drv_open() auf (über drv_call(DRV_OPEN)). Die Funktion d_open() muß nur dann explizit aufgerufen werden, wenn der Driver mehrere Devices verwaltet. Dann muß natürlich ein d_close() am Ende stehen.

9.1.4. d_close() - Driver schließen

Syntax:

CMS_RETURN d_close(CMS_HANDLE dhandle)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Der Driver ***dhandle*** wird geschlossen.

Falls für den Driver eines der Flags CMSDRV_NONSHARED oder CMSDRV_DEVICES gesetzt ist, ruft CMS die Driver-Funktion drv_close() auf (über drv_call(DRV_CLOSE)).

9.1.5. d_handle() - Nächstes Driver-Handle liefern

Syntax:

CMS_HANDLE d_handle(CMS_HANDLE dhandle)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN

Return:

CMS_ HANDLE	> 0	nächstes Driver- Handle
	= 0	keine weiteren Driver- Handles vorhanden

Beschreibung:

Diese Funktion liefert zu einem gültigen Driver-Handle das nächste gültige Driver-Handle. Für ***dhandle*** = 0 wird das erste Driver-Handle geliefert, für das letzte gültige Driver-Handle wird 0 zurückgeliefert. Für Driver mit mehreren Devices wird ein Handle für jedes geöffnete Device geliefert.

9.1.6. d_call() - Driver-Funktion aufrufen

Syntax:

int d_call(CMS_HANDLE dhandle, int fun, STACK * parm)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN
int	fun	Funktionsnummer	IN
STACK *	parm	Aufrufparameter	IN

Return:

int	CMS_OK	Aufruf erfolgreich
	CMS_BADHANDLE	Ungültiges Handle
	CMS_FUNCTION	Ungültige Funktionsnummer
	andere < 0	Returncode der Driver-Anwendungsfunktion

Beschreibung:

Mit Hilfe von d_call() wird eine Funktion im Driver ***dhandle*** aufgerufen. Die Selektierung der Funktion erfolgt über die Funktionsnummer ***fun***. Die Zuordnung der Nummern zu Funktionen ist Driver-spezifisch; es gibt allerdings Standard-Funktionen, die über Standardnummern erreicht werden (s. Standard-Driver-Funktionen), z.B. die Nummern DRV_OPEN, DRV_CLOSE, DRV_READ, DRV_WRITE. ***parm*** gibt die Adresse der aktuellen Aufrufparameter an, die der Driver-Funktion übergeben werden sollen.
Der Rückgabewert ist gleich dem Rückgabewert der gerufenen Driverfunktion bzw. einem Errorcode.

9.1.7. d_getcb() - Driver Control Block abfragen

Syntax:

CMS_RETURN d_getcb(CMS_HANDLE dhandle, CMS_DCB ** dcb)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN
CMS_DCB **	dcb	Feld für Adresse des Kontrollblocks	OUT

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle

Beschreibung:

Es wird ein Zeiger auf den Driver-Kontrollblock von ***dhandle*** zurückgegeben. Der Kontrollblock darf nicht verändert werden !
Er hat folgenden Aufbau:

```
typedef struct                          /* driver control block      */
{
   ULONG devopen0, devopen32;           /* device open flags 64 bits */
   CMS_DRVENTRY call;                   /* call address              */
   CMS_REDIR ** red;                    /* ptr to redirection list   */
   void * inidata;                      /* ptr to initial data segm. */
   void * data[MAXPROC];                /* data segs for every proc. */
   unsigned int datalen;                /* length of data segment    */
   int devs;                            /* number of devices         */
   int flags;                           /* driver flags              */
   char name[NAMELEN];                  /* driver name               */
} DCB;
```

9.1.8. d_list() - Driver-Informationen ausgeben

Syntax:

void d_list(void)

Parameter:

keine

Return:

kein

Beschreibung:

Es wird eine Liste mit Informationen zu allen aktiven Drivern ausgegeben. Sie hat die Form:

```
DRIVER LIST:
 handle name     flags devs  dataptr   datalen
 $07000 d2       $0009    0  $34080000    5504
```

In einzelnen bedeuten die Parameter:

handle	Handle des Drivers
name	Name des Drivers
flags	Konfigurationsflags
devs	Anzahl der Devices
dataptr	Adresse des Datenbereichs
datalen	Länge des Datenbereichs

9.1.9. d_monitor() - Interne Driver-Daten ausgeben

Syntax:

CMS_RETURN d_monitor(CMS_HANDLE dhandle)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle
	andere < 0	Returncode von xyz_monitor()

Beschreibung:

Diese Funktion gibt den internen Zustand des Drivers ***dhandle*** aus, indem drv_monitor() aufgerufen wird (Das Flag CMSDRV_MONITOR muß gesetzt sein.). Die Art und Form der Ausgaben hängt vom jeweiligen Driver ab.

9.1.10. d_read() - Vom Driver lesen

Syntax:

int d_read(CMS_HANDLE dhandle, char * buf, int buflen, CMS_TIME timeout)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN
char *	buf	Daten-Puffer	OUT
int	buflen	maximale Pufferlänge	IN
CMS_TIME	timeout	Wartezeit in Millisekunden	IN

Return:

int	>= 0	Anzahl der gelesenenen Bytes
	CMS_BADHANDLE	Ungültiges Handle
	CMS_SIZE	Buffer für Message zu klein
	CMS_TIMEOUT	Wartezeit abgelaufen, kein Erfolg
	CMS_EMPTY	keine Message vorhanden
	andere < 0	Returncode von xyz_read()

Beschreibung:

CMS ruft die Driver-Funktion drv_read() auf (über drv_call(DRV_READ)) und holt damit Daten vom Driver ab. Die Daten werden in den Puffer ***buf*** geschrieben, die Anzahl der gelesenen Bytes wird als Returncode zurückgegeben. Falls keine Daten vorliegen, wird der aufrufende Prozeß solange suspendiert, bis der Driver Daten hat oder ***timeout*** msec abgelaufen sind.

9.1.11. d_look() - Daten vom Driver "ansehen"

Syntax:

int d_look(CMS_HANDLE dhandle, char * buf, int buflen)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN
char *	buf	Daten-Puffer	OUT
int	buflen	maximale Pufferlänge	IN

Return:

int	>= 0	Anzahl der gelesenenen Bytes
	CMS_BADHANDLE	Ungültiges Handle
	CMS_EMPTY	keine Message vorhanden
	andere < 0	Returncode von xyz_look()

Beschreibung:

d_look() entspricht d_read() mit dem Unterschied, daß in d_look() die Daten im internen Buffer des Drivers erhalten bleiben. Das bedeutet, ein nachfolgendes d_read() liefert dieselben Daten. CMS ruft die Driver-Funktion drv_look() auf (über drv_call(DRV_LOOK)).

9.1.12. d_write() - In Driver schreiben

Syntax:

int d_write(CMS_HANDLE dhandle, char * buf, int buflen, CMS_TIME timeout)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN
char *	buf	Daten-Puffer	IN
int	buflen	maximale Pufferlänge	IN
CMS_TIME	timeout	Wartezeit in Millisekunden	IN

Return:

int	>= 0	Anzahl der geschriebenen Bytes
	CMS_BADHANDLE	Ungültiges Handle
	CMS_SIZE	Datenn passen nicht in Zielpuffer
	CMS_TIMEOUT	Wartezeit abgelaufen, kein Erfolg
	andere < 0	Returncode von xyz_write()

Beschreibung:

CMS ruft die Driver-Funktion drv_write() auf (über drv_call(DRV_WRITE)) und übergibt damit dem Driver die Daten aus ***buf*** (***buflen*** Bytes). Die Anzahl der geschriebenen Bytes wird als Returncode zurückgegeben. Falls der Driver keine Daten entgegen nimmt, wird der aufrufende Prozeß solange suspendiert, bis der Driver die Daten nimmt oder ***timeout*** msec abgelaufen sind.

9.1.13. d_flush() - Driver leeren

Syntax:

CMS_RETURN d_flush(CMS_HANDLE dhandle)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle
	andere < 0	Returncode von xyz_flush()

Beschreibung:

CMS ruft die Driver-Funktion drv_flush() auf (über drv_call(DRV_FLUSH)) und veranlaßt damit den Driver, seine internen Puffer zu löschen.

9.1.14. d_control() - Driver-Umlenkung definieren

Syntax:

CMS_HANDLE d_control(CMS_HANDLE dhandle, CMS_HANDLE chandle,
CMS_CONTROL * icontrol, CMS_HANDLE * dest)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN
CMS_HANDLE	chandle	Control-Handle	IN
CMS_CONTROL *	icontrol	Datenstruktur mit den Umlenkungsparametern	IN
CMS_HANDLE *	dest	Liste von Zielinstanzen	IN

Return:

CMS_HANDLE	> 0	neues Control -Handle
	CMS_FULL	zu viele Redirection-Anweisungen
	CMS_NOMEM	kein Speicher mehr frei
	CMS_ERROR	anderer Fehler (zu viele Filterbedingungen, Ziele o.ä.)

Beschreibung:

Mit der Funktion d_control() werden die IPC-Umleitungen für den Driver ***dhandle*** definiert. Damit wird CMS veranlaßt, nachfolgende Daten, die ein Prozeß oder anderer Driver über d_write() an den Driver ***dhandle*** sendet, selektiv und vom Aufrufer unbemerkt umzulenken. Als neue Ziele können Queues, Pipes, andere Driver oder ein anderes Device desselben Drivers angegeben werden.

Die anderen Parameter entsprechen denen von q_control() und sind dort ausführlich erläutert, ebenso die Wirkung der Control-Struktur. Es wird keine Driver-Funktion aufgerufen.

9.1.15. d_signal() - Driver-Signal definieren

Syntax:

CMS_RETURN d_signal(CMS_HANDLE dhandle, int kind, CMS_SIGNAL signal)

Parameter:

CMS_HANDLE	dhandle	Driver-Handle	IN
int	kind	Art des Ereignisses	IN
CMS_SIGNAL	signal	Signalwort	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	Ungültiges Handle
	andere < 0	Returncode von xyz_signal()

Beschreibung:

CMS ruft die Driver-Funktion drv_signal() auf (über drv_call(DRV_SIGNAL)) und teilt dem Driver die für eine Signalisierung nötigen Daten mit. Die eigentliche Signalisierung übernimmt der Driver (siehe drv_signal()).

9.2. Standard-Driver-Funktionen

Im folgenden Text steht XYZ für einen beliebigen Driver, xyz-...() für die Export-Funktionen dieses Drivers. Die Einsprungfunktion eines beliebigen Drivers unter CMS heißt immer drv_call().

Die Funktionen xyz_init(), xyz_exit() und drv_call() müssen in allen Drivern vorhanden sein, da sie von den entsprechenden Driver-Management-Funtkionen automatisch aufgerufen werden. Von allen Drivern im Nonshared Data Model sowie von allen Drivern, die mehrere Devices unterstützen, werden die Standard-Funktionen xyz_open() und xyz_close() benötigt. In diesen Drivern sind die Flags CMSDRV_NONSHARED oder CMSDRV_DEVICES gesetzt.

Alle Communication Driver (Flag CMSDRV_IPC gesetzt) müssen die sechs Kommunikationsfunktionen implementieren (xyz_read(), xyz_look(), xyz_write(), xyz_flush(), xyz_signal()).

Die Funktion xyz_monitor() ist optional für alle Driver; das Flag CMSDRV_MONITOR ist entsprechend zu setzen.

Zu jeder dieser Standardfunktionen gehört eine ebenfalls standardisierte Funktionsnummer (xyz_init() entspricht DRV_INIT usw.); andere Driverfunktionen sollen diese Nummern nicht als Funktionsnummern benutzen.

Die Funktion drv_call() ist die Entry-Funktion des Drivers, die CMS mit d_create() mitgeteilt wird. **Alle** Driver-Aufrufe erfolgen über drv_call() und verzweigen dann erst über die Funktionsnummer.

Die Standard-Driver-Funktionen xyz_write(), xyz_read() usw. können zwar direkt aufgerufen werden. Besser ist es jedoch, die CMS-Funktionen d_write(), d_read() usw. zu verwenden, die ihrerseits auf die Standard-Driver-Funktionen abgebildet werden. Hierbei können nämlich die Features von CMS genutzt werden (transparente Umlenkung bei Bedarf, Nutzung des Timeout, wenn der Aufruf blockiert). Natürlich muß zunächst mit d_open() das Handle geholt werden.

9.2.1. xyz_init() - Driver global initialisieren

Syntax:

CMS_RETURN xyz_init(int argc, char ** argv)

Parameter:

int	argc	Anzahl der Kommandozeilenargumente	IN
char **	argv	Zeiger auf Liste der Kommandozeilenargumente	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	< 0	Fehler lt. Festlegung des Drivers

Beschreibung:

Diese Funktion muß in jedem Driver vorhanden sein. Hier nimmt er die globale Initialisierung der Daten vor.

xyz_init() wird von CMS innerhalb von d_create() aufgerufen. Dabei werden auch die Kommandozeilenargumente übergeben. Falls xyz_init() einen negativen Returncode zurückliefert, wird das als Errorcode gewertet und der Driver wird nicht erzeugt.

9.2.2. xyz_exit() - Driver global deinitialisieren

Syntax:

CMS_RETURN xyz_exit(void)

Parameter:

keine

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	falsches Driver-Handle
	andere < 0	Fehler lt. Festlegung des Drivers

Beschreibung:

Diese Funktion muß in jedem Driver vorhanden sein. Hier nimmt er die globale Deinitialisierung bzw. Freigabe seiner Daten vor.
xyz_exit() wird von CMS innerhalb von d_delete() aufgerufen.

9.2.3. drv_call() - Driver-Funktion aufrufen

Syntax:

CMS_RETURN drv_call(CMS_HANDLE phandle, int dev, int fun, STACK * parm)

Parameter:

CMS_HANDLE	phandle	Handle des rufenden Prozesses	IN
int	dev	Gerätenummer	IN
int	fun	Funktionsnummer	IN
STACK *	parm	Aufrufparameter der Funktion ***fun***	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	CMS_BADHANDLE	falsches Driver-Handle
	andere < 0	Fehler lt. Festlegung des Drivers

Beschreibung:

drv_call() ist die Entry-Funktion des Drivers, deren Adresse bei d_create() angegeben wurde.
Alle Driver-Aufrufe erfolgen über diese Funktion !

phandle ist das Handle des aufrufenden Prozesses, ***dev*** ist die Devicenummer.
fun ist die Funktionsnummer, ***parm*** zeigt auf die Parameterliste.

Die Funktion drv_call() verzweigt mittels der Funktionsnummer auf die einzelnen Driver-Funktionen. Die jeweiligen Parameter können dabei mit Hilfe des GETPARM-Makros (s.u.) von der Parameterliste geholt werden, z.B. werden die Parameter, die bei den anderen Standard-Driver-Funktionen angegeben sind, durch ***parm*** übermittelt. Es kann zusätzlich sinnvoll sein, der jeweiligen Driver-Funktion auch noch ***phandle*** und/oder ***dev*** zu übergeben !

Der Returncode der Driver-Funktion wird auch als Returncode von drv_call() zurückgegeben.
Für eine ungültige Funktionsnummer muß von drv_call() ein (negativer) Errorcode zurückgeliefert werden.

Die Standard-Driver-Funktionen werden im mitgelieferten Module drvmain.c aufgerufen. Der Entwickler eines Drivers braucht sich darum nicht zu kümmern, sondern muß nur die zusätzlichen Export-Funktionen des Drivers nach demselben Muster einbinden (siehe Kapitel “3.2. Driver”).

9.2.4. xyz_open() - Prozeß- oder Devicedaten initialisieren

Syntax:

CMS_RETURN xyz_open(int dev, CMS_HANDLE phandle)

Parameter:

int	dev	Gerätenummer	
CMS_HANDLE	phandle	Prozeß-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	< 0	Fehler laut Festlegung des Drivers

Beschreibung:

Diese Funktion muß in jedem Driver mit Nonshared Data sowie in jedem Driver mit mehreren Devices vorhanden sein, da sie automatisch beim ersten Driver-Aufruf aus einem Prozeß aktiviert wird, d.h. der erste Aufruf führt zu einem d_open() und darin wird xyz_open() aufgerufen. ***phandle*** ist das Handle des aufrufenden Prozesses, ***dev*** ist die Devicenummer.

In xyz_open() hat der Driver Gelegenheit, sein Datensegment oder nur die Device-spezifischen Daten zu initialisieren. Im Falle Nonshared Data steht ihm für jeden Aufrufer ein eigenes Datensegment zur Verfügung. Die Umschaltung und Verwaltung wird von CMS vorgenommen

Im Shared-Data-Model muß der Driver in xyz_open() und allen weiteren Funktionen die Zuordnung der Datenbereiche zu den Aufrufern mittels des Parameters ***dev*** selbst überwachen (z.B. durch geeignete Indizierung).

Falls xyz_open() einen negativen Returncode zurückliefert, wird das als Errorcode gewertet und der Driver wird nicht geöffnet, d.h. der Prozeß kann ihn nicht aufrufen.

9.2.5. xyz_close() - Prozeß- oder Devicedaten deinitialisieren

Syntax:

CMS_RETURN xyz_close(int dev, CMS_HANDLE phandle)

Parameter:

int	dev	Gerätenummer	IN
CMS_HANDLE	phandle	Prozeß-Handle	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	< 0	Fehler laut Festlegung des Drivers

Beschreibung:

Diese Funktion muß in jedem Driver mit Nonshared Data sowie in jedem Driver mit mehreren Devices vorhanden sein. Der Driver erhält Gelegenheit zum "Aufräumen". xyz_close() wird von CMS innerhalb von d_close() aufgerufen. ***phandle*** ist das Handle des aufrufenden Prozesses, ***dev*** ist die Devicenummer.

Bei einem Driver im Nonshared Data Model wird der Returncode von xyz_close() beachtet: Wird CMS_OK zurückgegeben, dann ist der Driver "fertig mit dem Aufräumen", und CMS gibt das Driver-Datensegment für diesen Prozeß wieder frei. Wenn das Datensegment noch benötigt wird, etwa weil noch andere Devices vom selben Prozeß aktiv sind, dann sollte der Driver CMS_EXIST zurückgeben.

9.2.6. xyz_read() - Daten lesen

Syntax:

int xyz_read(int dev, char * buf, int buflen)

Parameter:

int	dev	Gerätenummer	IN
char *	buf	Datenpuffer	OUT
int	buflen	Länge des Datenpuffers	IN

Return:

int	> 0	Anzahl der gelesenen Bytes
	= 0	keine Daten vorhanden
	< 0	Fehler laut Festlegung des Drivers

Beschreibung:

Diese Funktion muß in jedem Communication Driver vorhanden sein.

Mit xyz_read() können Daten vom Driver abgeholt werden. Sie werden in den Buffer ***buf*** (Bufferlänge ***buflen***) kopiert. Die Anzahl der in den Buffer kopierten Bytes wird als Returncode zurückgegeben. Falls keine Daten verfügbar sind, muß der Driver 0 zurückgeben, d.h. der Driver sollte nicht auf die Daten warten. Ein negativer Returncode wird als Errorcode interpretiert.

Falls ein wartendes Lesen erwünscht ist, muß statt dieser die Funktion d_read() aufgerufen werden. Sie greift auf xyz_read() zurück, enthält aber außerdem den Wartemecahnismus.

Innerhalb von xyz_read() dürfen keine CMS-Funktionen mit Ausnahme von p_send() aufgerufen werden !

9.2.7. xyz_look() - Daten "ansehen"

Syntax:

int xyz_look(int dev, char * buf, int buflen)

Parameter:

int	dev	Gerätenummer	IN
char *	buf	Datenpuffer	OUT
int	buflen	Länge des Datenpuffers	IN

Return:

int	> 0	Anzahl der gelesenen Bytes
	= 0	keine Daten vorhanden
	< 0	Fehler laut Festlegung des Drivers

Beschreibung:

Diese Funktion muß in jedem Communication Driver vorhanden sein.

xyz_look() entspricht xyz_read() mit dem Unterschied, daß in xyz_look() die Daten im driver-internen Puffer erhalten bleiben. Das bedeutet, ein nachfolgendes xyz_read() liefert dieselben Daten.

9.2.8. xyz_write() - Daten schreiben

Syntax:

int xyz_write(int dev, char * buf, int buflen)

Parameter:

int	dev	Gerätenummer	IN
char *	buf	Datenpuffer	IN
int	buflen	Länge des Datenpuffers	IN

Return:

int	> 0	Anzahl der geschriebenen Bytes
	= 0	Driver kann keine Daten aufnehmem
	< 0	Fehler laut Festlegung des Drivers

Beschreibung:

Diese Funktion muß in jedem Communication Driver vorhanden sein.

Mit xyz_write() können Daten an den Driver übergeben werden. Es werden ***buflen*** Bytes ab der Adresse ***buf*** übergeben und vom Driver in einen Driver-internen Puffer übernommen. Die Anzahl der übernommenen Bytes wird als Returncode zurückgegeben.

Der Driver soll alles oder nichts verarbeiten: Falls der Driver nicht alle Daten übernehmen kann, muß er 0 zurückgeben, d.h. der Driver sollte nicht auf das Freiwerden des Puffers warten. Ein negativer Returncode wird als Errorcode interpretiert.

Innerhalb von xyz_write() dürfen keine CMS-Funktionen mit Ausnahme von p_send() aufgerufen werden !

Wartendes Schreiben kann mit d_write() realisiert werden.

9.2.9. xyz_flush() - Buffer leeren

Syntax:

CMS_RETURN xyz_flush(int dev)

Parameter:

int	dev	Gerätenummer	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	< 0	Fehler laut Festlegung des Drivers

Beschreibung:

Diese Funktion muß in jedem Communication Driver vorhanden sein.

Die driver-internen Read- und Write-Buffer für das angegebene Device werden gelöscht.

9.2.10. xyz_signal() - Driver-Signal definieren

Syntax:

CMS_RETURN xyz_signal(int dev, CMS_HANDLE phandle, int kind, CMS_SIGNAL signal)

Parameter:

int	dev	Gerätenummer	IN
CMS_HANDLE	phandle	Prozeß-Handle	IN
int	kind	Art des Ereignisses	IN
CMS_SIGNAL	signal	zu versendenden Signalwort	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	< 0	Fehler laut Festlegung des Drivers

Beschreibung:

Diese Funktion muß in jedem Communication Driver vorhanden sein.

Bei Eintreten des mit ***kind*** spezifizierten Ereignisses soll der Driver an den Prozeß ***phandle*** das Signal ***signal*** senden. Dieses Verhalten muß solange bestehen bleiben, bis es durch erneuten Aufruf von xyz_signal() mit demselben ***phandle*** wieder geändert wird.

Ein ***signal*** == 0 bedeutet: Für diesen Prozess soll keine Signalisierung (mehr) erfolgen. Ein ***signal*** != 0 muss mit einem schon vorhandenen Signalwort für das gleiche Ereignis und den gleichen Prozess 'verodert' werden !

Folgende Ereignisse sollte der Driver signalisieren können:

kind = CMS_CANREAD : Es können Daten gelesen werden.
kind = CMS_CANWRITE : Es können Daten geschrieben werden (wenigstens 1 Byte).

Zum Signalisieren steht dem Driver die Funktion p_send() zur Verfügung, die einzige CMS-Funktion, die innerhalb der Standard-Driver-Funktionen xyz_read() und xyz_write() aufgerufen werden darf.

9.2.11. xyz_monitor() - Interne Driver-Daten ausgeben

Syntax:

CMS_RETURN xyz_monitor(int dev)

Parameter:

int	dev	Gerätenummer	IN

Return:

CMS_RETURN	CMS_OK	Funktion erfolgreich
	< 0	Fehler laut Festlegung des Drivers

Beschreibung:

Diese Driver-Funktion ist optional. Sie muß vorhanden sein, wenn das Flag CMSDRV_MONITOR gesetzt ist. Falls sie vorhanden ist, soll sie den internen Zustand des Drivers in geeigneter Form ausgeben.

10. VCMS für Windows NT 4.0

In diesem Abschnitt werden die Besonderheiten des “Virtuellen” CMS für Windows NT 4.0 beschrieben.

10.1. Unterstützte Funktionen (Stand 08.03.2015)

In der folgenden Tabelle bedeuten:

- ja: Die Funktion ist implementiert, wie in diesem Handbuch beschrieben.
- nein: Die Funktions wird nicht implementiert.
- dummy: Die Funktion kann aufgerufen werden, das bewirkt aber nichts.
  P: Die Funktion darf nur in einem CMS-Prozess aufgerufen werden (also z.B. nicht in main()).

Funktion	**in VCMS**	**Bemerkungen**
s_time()	ja	Es wird die Zeit seit dem 1.1.1970 in msec geliefert.
s_settime()	nein	
s_mode()	ja	VCMS etabliert kein eigenes Scheduling und arbeitet daher (wie NT) immer im Modus TIMESLICE. s_mode() verstellt die Ausgabe-Modus (s.u.): Bit 7 des Parameters entspr. dem Flag 'i', die unteren Bits einem der Zeichen 'w', 'f', 's', 'n'.
s_mode2()	ja	Diese Funktion gibt es **NUR** in VCMS.
s_getcb()	ja	
s_list()	ja	Ausgabe ins CMS-Fenster
p_create()	ja	Als Priorität sollte CMS_NORMPRIO benutzt werden, als Stackgröße STDSTACK.
p_delete()	ja	p_delete() funktioniert nur, wenn es innerhalb des NT-Prozesses aufgerufen wird, in dem auch p_create() für diesen CMS-Prozess aufgerufen wurde (der Prozessname wird allerding immer gelöscht !). p_delete() löscht alle Timer, die dieser Prozess gestartet hat.
p_open()	ja	
p_close()	dummy	
p_handle()	dummy	
p_start()	ja	p_start() funktioniert nur, wenn es innerhalb des NT-Prozesses aufgerufen wird, in dem auch p_create() für diesen CMS-Prozess aufgerufen wurde.
p_stop()	dummy	
p_enable()	nein	
p_disable()	nein	
p_priority()	dummy	
p_schedule()	ja	bedient Windows-Messagequeue des CMS-Fensters. Wenn der letzte in einem Programm erzeugte CMS-Prozess gelöscht wurde, dann

		beendet der nächste p_schedule-Aufruf das Programm.
p_pid() **P**	ja	Makro; nicht in Timer-Callback-Funktion verwenden
p_sleep()	ja	
p_wait() **P**	ja	
p_send()	ja	
p_getset() **P**	ja	
p_getcb()	ja	
p_list()	ja	Ausgabe ins CMS-Fenster
t_create()	ja	
t_delete()	ja	
t_open()	ja	
t_close()	dummy	
t_handle()	dummy	
t_start() **P**	ja	Timer-Callback-Funktion läuft als eigener Thread concurrent zu den CMS-Prozessen. In der Timer-Callback-Funktion sollen keine Driver-Aufrufe erfolgen.
t_stop()	ja	
t_signal() **P**	ja	
t_status()	ja	
t_wait()	dummy	
t_getcb()	dummy	
t_list()	ja	Ausgabe ins CMS-Fenster
u_alloc()	ja	
u_free()	ja	
u_out()	ja	Ausgabe ins CMS-Fenster
u_puts()	ja	Ausgabe ins CMS-Fenster
u_route()	nein	
r_create()	ja	
r_delete()	ja	
r_open()	ja	
r_close()	dummy	
r_handle()	dummy	
r_request()	ja	
r_release()	ja	
r_signal() **P**	dummy	
r_status()	dummy	
r_getcb()	dummy	
r_list()	ja	Ausgabe ins CMS-Fenster
e_create()	nein	
e_delete()	nein	
e_open()	nein	
e_close()	nein	

e_handle()	nein	
e_set()	nein	
e_wait()	nein	
e_getset()	nein	
e_signal()	nein	
e_status()	nein	
e_getcb()	nein	
e_list()	nein	
q_create()	ja	kein Pointer-Modus
q_delete()	ja	
q_open()	ja	
q_close()	dummy	
q_handle()	dummy	
q_read()	ja	kein Pointer-Modus
q_look()	ja	
q_write()	ja	kein Pointer-Modus, noch keine Redirection
q_writeprio()	nein	
q_flush()	ja	
q_control()	ja	
q_signal() **P**	ja	
q_status()	ja	
q_getcb()	ja	
q_list()	ja	Ausgabe ins CMS-Fenster
m_create()	nein	
m_delete()	nein	
m_open()	nein	
m_close()	nein	
m_handle()	nein	
m_read()	nein	
m_look()	nein	
m_write()	nein	
m_flush()	nein	
m_control()	nein	
m_signal()	nein	
m_status()	nein	
m_getcb()	nein	
m_list()	nein	
d_create()	ja	Achtung: Die Aufrufe von Driver-Funktion werden NICHT serialisiert !
d_delete()	ja	
d_open()	ja	
d_close()	ja	
d_handle()	dummy	

d_call()	nein	
d_getcb()	dummy	
d_list()	ja	
d_monitor()	dummy	
d_read()	ja	
d_look()	ja	
d_write()	ja	
d_flush()	ja	
d_control()	ja	
d_signal() **P**	ja	

10.2. Error-Codes

Die Funktionen liefern in bestimmten Situationen zum Teil CMS-Error-Codes, die nicht in der Beschreibung der einzelnen Funktionen erwähnt sind. Diese werden später im Handbuch ergänzt. Als generelle Regel gilt, dass CMS-Error-Codes immer negativ sind. Alle vorkommenden Errorcode sind in cms.h als Konstanten definiert.

10.3. CMS-Ausgaben

Beim Aufruf von VCMS wird festgelegt, wohin über VCMS-Funktionen getätigte Ausgaben gelangen (siehe 10.7. Start und Ende). Dies betrifft alle Ausgaben über u_out(), u_puts() und die .._list()-Funktionen.

Es können mehrere Ausgabeziele gleichzeitig ausgewählt werden (z.B. Window und File).

Ausgaben in das VMCS-Fenster ("cms –w) werden nur angezeigt, wenn die Windows-Messagequeue dieses Fensters ausgelesen wird. Das geschieht, indem regelmäßig p_schedule() aufgerufen wird.

Standardmäßig wird das main()-Programm einer CMS-Anwendung mit dem RETURN-Makro beendet. Diese Makro beendet in der NT-Version das Programm nicht sondern enthält eine Endlosschleife mit p_schedule()-Aufrufen, wodurch das CMS-Fenster korrekt bedient wird.

10.4. Control-Blöcke

Der Inhalt der Control-Blöcke unterscheidet sich von dem für das “reine” CMS definierten. Aus CMS_SCB dürfen nur `major` und `minor` verwendet werden, aus allen anderen Control-Blöcken nur `name[]`.

10.5. Driver mit Kommandozeilenargumenten

In VCMS wird drv_init() automatisch aufgerufen, aber mit argc == 0. Um hier Kommandozeilenargumente übergeben zu können, muss man in einem main()-Programm drv_init() explizit aufrufen. Im einfachsten Fall:

```
void main( int argc, char ** argv )
{
CMS_RETURN rc;

   rc = xyz_init( argc, argv );
   if( rc < 0 )
      exit( 1 );

   RETURN;
}
```

10.6. Übersetzen

Die CMS-Definitionen werden mittels `#include "cms.h"` eingebunden. Der Include-Pfad dafür muss ..*include*,..*include*_*nt* lauten.
Es werden dann folgende H-Files eingezogen:
..\include_nt\cms.h ..\include_nt\cmstypes.h ..\include\int64.h ..\..\sema.h

Zum Linken ist die Bibliothek CMS.lib erforderlich, diese befindet sich in .._*nt**lib**debug* (die Debug-Version).

10.7. Start und Ende

Der Start von CMS erfolgt automatisch durch den Start der ersten CMS-Applikation. Alternativ kann CMS auch mittels CMS.EXE gestartet werden. In diesem Fall können folgende Parameter übergeben werden:

```
cms [-(w|f|s|n)] [{iptrqd} [sec]]
cms [-n | {-w|-s|-f[File]}] [{iptrqd} [sec]]

-n        keine Ausgaben
-w        Ausgaben an das Window "cms output"
-fFile    Ausgaben an das angegebene File (Default: "output.cms")
-s        Ausgaben an stdout

Der erste Parameter gibt an, wohin die CMS-Ausgaben erfolgen:
-w   Window (Default)
-f   File 'output.cms'
-s   stdout
-n   none
```

Wenn (weitere) Parameter angegeben sind, dann werden im CMS-Fenster periodisch Listen der CMS-Objekte angezeigt (p = Prozesse, t = Timer, r = Semaphoren, q = Queues, d = Driver). *sec* ist die Periodendauer in Sekunden (Default: 10 sec).
Der Parameter 'i' bewirkte die Ausgabe der aktuellen Prozess- und Thread-Anzahl bei jedem Prozess- (oder Thread-)Start oder Ende.

Beim Beenden einer CMS-Applikation werden alle CMS-Objekte (ausser Driver), die in diesem Programm erzeugt wurden, wieder gelöscht. Driver werden gelöscht, sobald das letzte sie benutzende Programm beendet wird.

Wenn die letzte CMS-Applikation (einschließlich CMS.EXE) beendet wird, entfernt sich CMS aus dem Speicher. Damit sind alle CMS-Objekte gelöscht.

10.8. Beispiele

Im Verzeichnis q2ab befindet sich eine Beispiel-Applikation. Es werden zwei Programme erzeugt: q2b erzeugt eine Queue und liest sie aus, q2a öffnet diese Queue und schreibt in sie.

Für eigene Experimente soll dieses Verzeichnis auf den eigenen PC kopiert werden !

Das Makefile in diesem Verzeichnis kann als Vorbild verwendet werden.

Der Start kann im Verzeichnis .._nt\bin z.B. mit
```
start q2a
q2b
```
erfolgen.

Im Verzeichnis d3 befindet sich ein Beispiel-Driver (d3) mit Anwendung (d3test). Hier gilt sinngemäß das Gleiche wie oben. Start mit
```
d3test
```

Im Verzeichnis sample.cms sind weitere (nicht unbedingt selbsterklärende) Beispiel-Applikationen enthalten (alte Testprogramme für CMS).

10.9. Implementierung

VCMS ist als DLL implementiert. Die einzelnen CMS-"Objekte" werden wie folgt auf Win32-"Objekte" abgebildet:

CMS	**NT**
Process	Thread
Timer	mit eigenem Thread und Sleep() [keiner der Win32-Timer ist zu gebrauchen]
Semaphore	Interlocked variables
Queue	Memory Mapped File
Driver	DLL