I N S T I T U T F Ü R I N F O R M A T I K
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN
Lehrstuhl Echtzeitsysteme und Robotik
Dr. Gerhard Schrott

Programmier-Praktikum Echtzeitprogrammierung

Inhaltsverzeichnis

0 Ada-Entwicklungsumgebung

Auf den Rechnern lynsiegert5-10 im Raum O243 ist das Echtzeitbetriebssytem LynxOS, eine UNIX-Variante mit UNIX-Schnittstellen installiert. Unter LynxOS steht die Ada 83 Entwicklungsumgebung von Aonix zur Verfügung. Diese umfaßt den Compiler, den Binder und weitere Tools. Die Umgebung wird mit dem Befehl `ada` aufgerufen und meldet sich mit dem Prompt `Ada.` Die Umgebungsvariable `ALSYCOMP_DIR` muß auf `/usr/local/DIR/Ada-5.5.9` gesetzt werden. Am besten kopieren Sie sich /usr/local/LynxOS/ADA_LIBS/.cshrc in Ihr Homeverzeichnis.

Bei Ihrer ersten Sitzung müssen Sie folgende Kommandos ausführen. Die Kommandos können meist mit dem ersten Buchstaben abgekürzt werden. `/pfad/` steht für Ihren expliziten Pfadnamen:

`family_manager.new(Family=>/pfad/adafam)`	Einrichten einer eigenen Familie für die Ada-Libraries
`lib_man`	Aufruf Library Manager
`other(Family=>/pfad/adafam)`	Wechsel zur neuen Familie
`new(Library=>/pfad/adalib)`	Erzeugung einer neuen Library in dieser Familie
`quit`	
	Default Einstellungen
`default.lib_manager(Family=>/pfad/adafam)`	Verwenden der Familie adafam
`default.compile(library=>/pfad/adalib)`	Verwenden der Bibliothek adalib beim Übersetzen
`default.bind(library=>/pfad/adalib)`	Verwenden der Bibliothek adalib beim Binden
`default.unit(library=>/pfad/adalib)`	Verwenden der Bibliothek adalib beim Zugriff auf Objekte in der Biliothek
`def.comp(Memory=>1000,Stack=>32000)`	Default Speichergrößen
`def.comp(Error=>100)`	Abbruch der Übersetzung nach 100 Fehlern
`def.comp(text=>yes,warning=>no)`	Ausgabe Übersetzungsprotokoll
`default.quit(defaults=>keep)`	Alle Defaultwerte sind beim nächsten Aufruf der Ada Entwicklungsumgebung automatisch eingestellt.

Es ist sinnvoll, in einer Gruppe auf einer gemeinsamen Familie und auf gemeinsamen Bibliotheken zu arbeiten. Erzeugen Sie also die Familie und die Bibliotheken nur im Dateisystem eines Teilnehmers und tragen Sie diese bei allen anderen mit expliziter Pfadangabe ein.

Nun stehen u.a. folgende Befehle zur Verfügung:

`compile datei_name, library_name`
Compiliert die in datei_name enthaltenen Prozeduren und speichert sie in der Library library_name. Ein Übersetzungsprotokoll wird in die Datei `datei_name.lis` geschrieben. Wird kein library_name angegeben, so wird die default-Einstellung verwendet.

`bind prozedur_name, libary_name`
Bindet die Prozedur prozedur_name aus der Library library_name und erstellt für "Hauptprogramme" ein ausführbares Lynxosprogramm. Alle von der Prozedur verwendeten Unterprogramme müssen in der Library library_name enthalten sein. Ein Protokoll wird in die Datei `datei_name.MAP` geschrieben. Wird kein library_name angegeben, so wird die default-Einstellung verwendet.

`unit.list`
Zeigt alle Objekte in der momentan eingestellten library. Einstellung durch default oder `unit.other libary_name`.

`library.unlock library_name`
Der Zugriff auf die Library erfolgt zur Konsistenzerhaltung immer exklusiv. Sollte die Sperre nach einem Fehler nicht gelöst sein, so kann dies durch obiges Kommando erfolgen. Wird kein library_name angegeben, so wird die default-Einstellung verwendet.

Das ausführbare Programm kann entweder direkt unter LynxOS aufgerufen werden oder in der Ada-Umgebung durch `system.prozedur_name` gestartet werden. Wollen Sie ein Protokoll Ihrer Ausgabe auf eine Datei erhalten, so verwenden Sie die folgende pipe unter LynxOS: `prozedur_name | tee ausgabedatei`.

Die Ada-Umgebung *muß* durch `quit` verlassen werden, da sonst das Ada-System den Rechner blockiert.

Weitere Befehle und Optionen der Ada Entwicklungsumgebung finden Sie im verteilten Handbuch *LynxOS und die Programmiersprache ADA*. Manuale zu LynxOS und Ada 83 stehen im Regal in Raum O245. Auch im WWW sind Ada-Manuale und Programmierhandbücher zu finden, z.B.
`http://www.ada-europe.org/resources.html`
`http://www.informatik.uni-stuttgart.de/ifi/ps/ada-doc/rm83/`

Unter Ihrer Kennung ist auf den Rechnern hpsiegertx ein Directory eingerichtet und auf den LynxOS-Rechnern gemountet. Sie können auf Ihre Daten somit sowohl von den HP-Rechnern des Lehrstuhls als auch von den Pentium-Rechnern unter LynxOs zugreifen. Außerdem können Sie von den Rechner in der HP-Halle über remote login auf den LynxOS-Rechnern arbeiten. Sollte aus irgendwelchen Gründen eine korrekte Beendigung Ihrer Sitzung auf den LynxOS-Rechner nicht möglich sein (z.B. Verbindung bricht ab), so teilen Sie dies bitte per email an schrott@informatik oder baumanna@informatik (Systemverwalterin) mit. Sollte bei remote-login innerhalb der Ada-Umgebung delete nicht funktionieren, so versuchen Sie es mit CTRL-h.

Auf den LynxOs-Rechnern sind die Editoren vi und xedit verfügbar, auf den HP-Rechnern zusätzlich xemacs. Ihr Passwort können Sie auf den HP-Rechnern hpsiegert22-24 mit `passwd` ändern; um diese Änderung auch auf die LynxOS-Rechner zu übertragen, müssen Sie anschließend noch `pushpass` ausführen.

!Achtung!
Gehen Sie bitte vorsichtig und behutsam mit den Modellen um. Bei Hardware–Problemen informieren Sie bitte einen der Praktikumsbetreuer und versuchen nicht, die Probleme selbst zu beheben.

1 Erfolgreiche Teilnahme am Praktikum

Folgende Leistungen sind Voraussetzung für den Erwerb des Praktikumscheins:

a) Teilnahme an allen Praktikumsnachmittagen
(maximal zwei entschuldigte Abwesenheiten)

b) Die Aufgaben der Kapitel 2-6 müssen erfolgreich bearbeitet werden.

c) Eine Woche vor Praktikumsende ist pro Gruppe eine Ausarbeitung abzugeben, die für jede der bearbeiteten Aufgaben folgendes enthalten muß:
- Auf DIN A4-Format zugeschnittenes Programmlisting.
- Programmdokumentation.
- Beantwortung der in der Aufgabe gestellten Fragen.

d) Am Semesterende wird in einer einstündigen Klausur von jedem Teilnehmer die selbständige Programmierung einer kleinen Aufgabe geprüft.

2 Einführung

Diese Aufgabe soll in die Programmierumgebung von Ada einführen. Sie sollen das folgende fehlerhafte Ada-Programm verbessern, übersetzen, binden und ausführen. Der Quelltext dieses Programms ist in der Datei /usr/local/LynxOS/ADA_LIBS/ada0-f.ada enthalten, die Sie sich in Ihr Directory kopieren können.

Erläutern Sie kurz, wie das Programm aufgebaut ist und welche Funktion die einzelnen Programmkonstrukte haben.

Das Echtzeitbetriebssytem LynxOS führt die Prozessorzuteilung strikt präemptiv nach Prioritäten ohne Zeitscheiben durch. Der Ablauf des Programms wird also entscheidend durch die Wahl der Prioritäten der Tasks bestimmt. Testen Sie das Programm mit unterschiedlichen Prioritätenzuordnungen und erklären Sie den zeitlichen Ablauf des Programms (Zeitdiagramm).

Geben Sie das korrigierte Programmlisting und die kommentierte Programmausgabe der Aufgabe im DIN A4-Format geheftet ab.

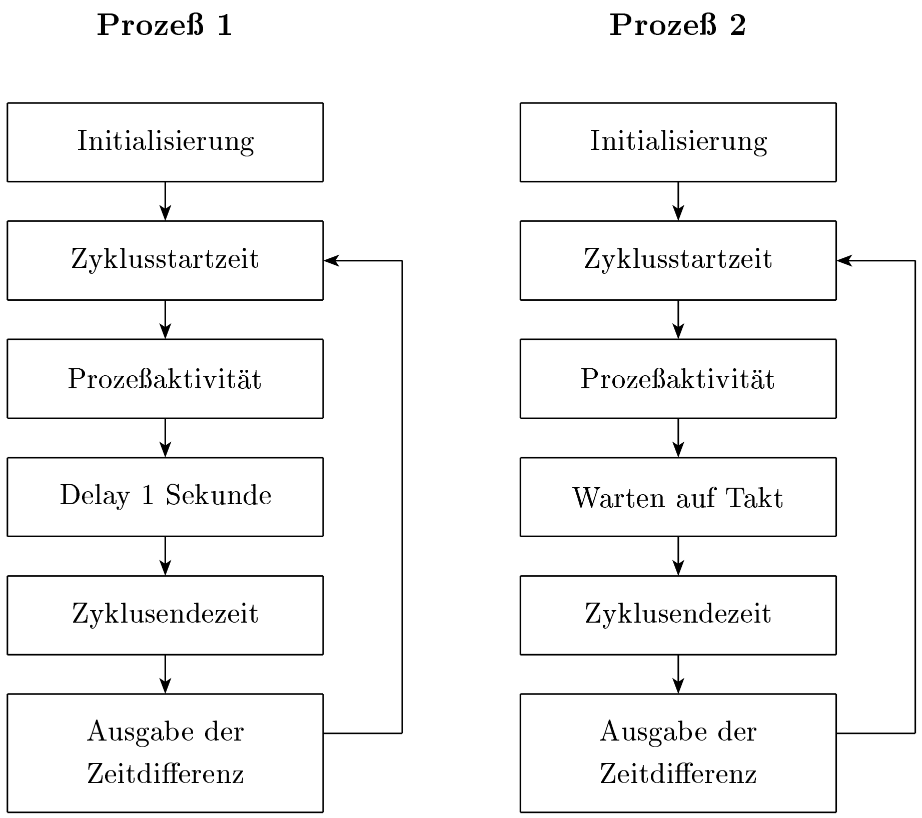
```
with text_io; use text_io;

procedure main is
pragma PRIORITY(50);     --Prioritaet der Task aus 1..253
                         --hoechste Prioritaet ist 253

package intout is new integer_io(integer);      --Inkarnation eines generischen
                                                --Package aus text_io
x : float := 0.0;
```

```
task ausgabe1 is
  pragma PRIORITY(40);
end ausgabe1;

task ausgabe2 is
  pragma PRIORITY(30);
end ausgabe2;

task body ausgabe1 is
begin
  put_line("Start ausgabe1");
  for i in 1..10 loop
    put(i); put_line(". Zeile aus Task ausgabe1");
    delay 1.0;
  end;
exception
  when others => put_line("ausgabe1 mit Fehler abgebrochen!");
end ausgabe1;

task body ausgabe2 is
begin
  put_line("Start ausgabe2");
  accept start;
  put_line("ausgabe2 nach accept start");

  for i in 1..5 loop
    put(i); put_line(". Zeile aus Task ausgabe2");
    delay 2.0;
  end loop
exception
  when others => put_line("ausgabe2 mit Fehler abgebrochen!");
end ausgabe

begin

put_line("Start des Hauptprogramms");
delay 2.0;
start;

put_line("Start Schleife");
for i in 1..1000000 loop
  x:= x*x/x;
end loop;
put_line("Ende Schleife");

exception
  when others => put_line("main mit Fehler abgebrochen!");
end main;
```

3 Zyklisch ablaufende Prozesse

3.1 Aufgabenstellung

Programmieren Sie nach dem unten angegebenen Muster einen zyklisch ablaufenden Prozeß nach zwei Arten. Der Prozeß 1 soll nach jeder Prozeßaktivität eine Sekunde lang verzögert werden. Der Prozeß 2 dagegen soll im Sekundentakt seine Prozeßaktivität wieder aufnehmen.



1. Interpretieren Sie schriftlich das unterschiedliche Zeitverhalten der beiden Prozesse. Um ein unverfälschtes Ergebnis zu erhalten, müssen die Prozesse nacheinander ablaufen!

2. Geben Sie das kommentierte Programmlisting der Aufgabe im DIN A4-Format geheftet ab; speichern Sie die Programmausgabe auf eine Datei und geben Sie auch einen Ausdruck dieser Datei mit ab.

3.2 Implementierungshinweise

Wählen Sie als Prozeßaktivität (Beschäftigung des Rechners) die folgende Befehlsfolge:

```
x := 3.14159;
for i in 1..N loop x := Sin(x*x/x); end loop;
```

N soll bei jedem Durchlauf von 0 ausgehend um jeweils 500 000 bis maximal 5 000 000 erhöht werden.

Mathematische Funktionen sind im package `generic_elementary_functions` enthalten. Dieses Paket muß für den Datentyp `float` instantiiert werden.

Zur Zeitdifferenzberechnung lesen Sie die Systemzeit mit der Funktion clock aus; diese wird im package `calendar` bereit gestellt:

```
zeit : time;      -- Vereinbarung der Variablen Zeit
   :
   :
zeit := clock;    -- Startzeitpunkt der Messung
   :
   :
put(integer((clock-zeit)*1000),5); -- Ausgabe der Zeitdifferenz
                                   -- in Millisekunden
```

Die Verzögerung von 1 sec wird mit dem Befehl `delay 1.0` erreicht.

Da es in Ada keine Anweisung für periodisches Starten von Prozessen gibt, muß der Sekundentakt durch folgendes Konstrukt simuliert werden:

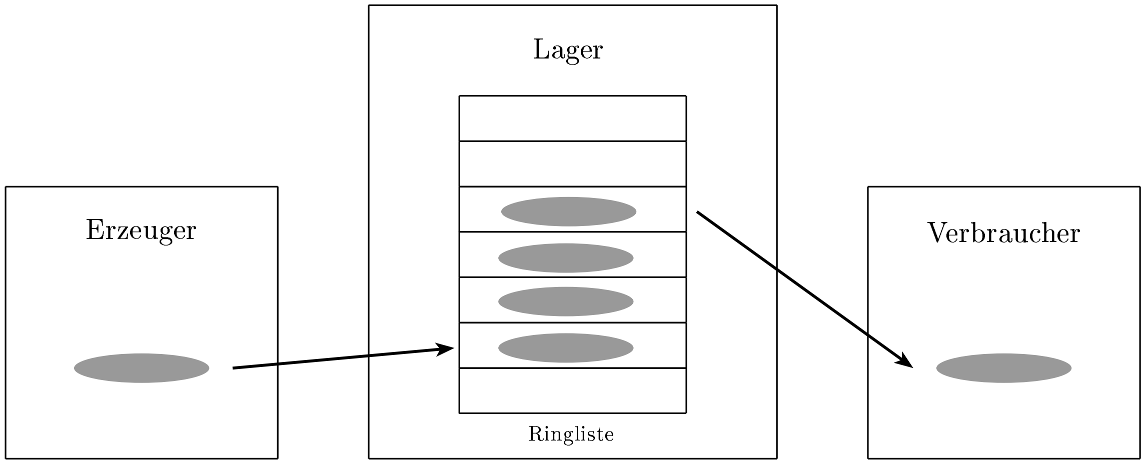
```
interval : duration:= 1.0;
periode  : time;           -- Vereinbarung der Variablen
   :
   :
periode := clock;          -- Startzeitpunkt der periodischen ktivitaet
   :
   :
periode := periode + interval;
delay periode - clock;
```

Hinweis: Bauen Sie in Ihre Ada-Programme Exceptionhandler analog zu Aufgabe 1 ein. Der Standard-Handler funktioniert nicht immer einwandfrei und Ihr Programm zeigt keinen Fehler an stürzt unkontrolliert ab.

4 Erzeuger-Verbraucher-Problem

4.1 Aufgabenstellung

Ein Erzeuger produziert Waren (Daten) und legt diese in einem Lager ab. Ein Verbraucher entnimmt die erzeugte Ware aus dem Lager in der gleichen Reihenfolge (FIFO) und verbraucht diese. Erzeuger und Verbraucher dürfen nicht gleichzeitig auf das Lager zugreifen. Außerdem ist die Lagerkapazität beschränkt.



Eine Lösung dieses Problems ist mit dem in Ada definierten Synchronisationskonzept Rendezvous möglich.

Lösung a: Schreiben Sie in einem Hauptprogramm (procedure) eine Task Lager, die den Ringpuffer mit den entries einlagern und auslagern realisiert, und je eine Task Erzeuger und Verbraucher, die darauf zugreifen.

Lösung b: Schreiben Sie ein Package Puffer, das wieder die Task lager enthält und Zugriffsprozeduren put und get nach außen verfügbar macht. Der Typ der Elemente im Ringpuffer und deren maximale Anzahl seien generisch. Testen Sie dieses Package in einem Hauptprogramm, welches wiederum die Tasks Erzeuger und Verbraucher beinhaltet.

Geben Sie das kommentierte Programmlisting der einzelnen Aufgaben im DIN A4-Format geheftet ab; speichern Sie die Ausgabe auf eine Datei und geben Sie auch einen Ausdruck dieser Datei mit ab.

4.2 Implementierungshinweise

Die Definition der Task Lager soll folgende Form haben:

```
task lager is
     entry ein(prod: in  integer);
     entry aus(prod: out integer);
end  lager;
```

Zur Implementierung des wechselseitigen Zugriffs der Funktionen `ein` und `aus` auf den Ringpuffer und zu ihrer Synchronisation bei leerem bzw. vollem Lager ist folgende bedingte select-Anweisung sinnvoll:

```
     select
        when <freie Plaetze im Ringpuffer vorhanden> =>
        accept ein(prod: in integer) do
          <trage Element in Ringpuffer ein>
        end;
     or
        when <Produkte im Ringpuffer gespeichert> =>
        accept aus(prod: out integer) do
           <hole Element aus Ringpuffer>
        end;
     or terminate;
     end select;
```

Protokollieren Sie jede Änderung im Lager.

Das Erzeugen und Verbrauchen von Produkten benötigt Bearbeitungszeit; fügen Sie deshalb an passender Stelle ein delay ein:

Erzeugen 0.6 Sekunden
Verbrauchen 1.0 Sekunden

Die Task Erzeuger soll 30 Produkte erzeugen und in das Lager mit 12 Plätzen ablegen. Die Task Verbraucher holt diese Produkte aus dem Lager und beendet sich, falls sie länger als 2 Sekunden auf Produkte warten muß.

Das generische Package Puffer soll folgende nach außen sichtbare Definitionen umfassen:

```
generic
   max: integer;
   type produkt is private;

package puffer is
   procedure put(x:produkt);
```

```
   function  get return produkt;
end puffer;
```

Das Package Puffer wird mit dem Ada-Compiler übersetzt und beim Binden des zugehörigen Hauptprogramms automatisch aus der Bibliothek geholt. Im Hauptprogramm wird eine Inkarnation von Puffer mit 10 Elementen vom Typ *teil* im Vereinbarungsteil erzeugt:

```
type teil is
   record
     nummer:  integer;
     gewicht: float;
   end record;

   package teilelager is new puffer(10,teil);
   use teilelager;
```

Die Task Verbraucher soll hier beendet werden, wenn die 30 erzeugten Produkte aus dem Lager geholt wurden.

5 Aufzugssteuerung

5.1 Aufgabenstellung

Für den im Praktikumsraum O243 dreifach aufgebauten Fischertechnik-Aufzug soll in ADA eine Hausmeistersteuerung implementiert werden:

Die Aufzugkabine läßt sich nur durch Anforderungen vom Innenraum der Kabine (Tastfeld) aus steuern. Die Kabine wartet immer bei offener Tür im letzten angefahrenen Stockwerk. Auf Tastendruck fährt die Kabine nach Schließen der Tür ins gewünschte Stockwerk. Anforderung für weitere Stockwerke werden nicht gespeichert.

Die Aufzugsmodelle werden über den in Kapitel 5.2 näher beschriebenen CAN-Bus von den LynxOS Rechner *lynsiegert6,7,10* gesteuert (siehe Abbildung 1). In jedem Aufzug verfügt jedes Stockwerk, das Tastfeld in der Kabine und der Kabinenmotor über einen eigenen Mikrocontroller MC68HC11 mit Anschluss an den CAN-Bus durch den CAN-Controller Intel 82526.

1. Befassen Sie sich mit dem in nächsten Kapitel beschrieben CAN-Bus und dem CAN-Bus Monitor. Verschicken Sie mit dem Monitor einfache Steuernachrichten, wie z.B. "rote Lampe im Erdgeschoß einschalten", und verfolgen Sie die Bus-Aktivitäten bei Betätigung von Tastern am Aufzug.
2. Studieren Sie die bereitgestellten Packages `canada.ada` sowie `aufzugdaten.ada`. Schreiben Sie mit deren Hilfe eine Sende-Prozedur, welche eine Botschaft über den CAN-Bus verschickt. Testen Sie diese Prozedur durch einfache Steueraufgaben und beobachten Sie die CAN-Bus-Aktionen mit dem Monitor.
3. Schreiben Sie eine Empfangs-Task, welche Botschaften des CAN-Bus empfängt, dekodiert und verwaltet. Realisieren Sie den Endetaster zum gezielten Beenden Ihres Programms.
4. Vervollständigen Sie Ihr bisheriges Programm, so daß die oben genannte Funktionalität einer Hausmeistersteuerung erreicht wird.
5. Führen Sie Ihr Programm vor.
6. Geben Sie das kommentierte Programmlisting im DIN A4-Format geheftet ab.

5.2 Der CAN-Bus

CAN (Controller Area Network) ist ein serielles Bussystem, das ursprünglich für den Einsatz in Kraftfahrzeugen entwickelt wurde, aber inzwischen aufgrund seiner Eigenschaften auch in der industriellen Automatisierungstechnik immer stärker eingesetzt wird. In KFZs gehen seine Einsatzgebiete von einfachen Steuerungen, wie z.B. der Scheibenwaschanlage bis hin zum Motormanagement, ABS oder der Katalysatorsteuerung.

In den internationalen Standardvorschlägen ISO/DIS 11898 und ISO/DIS 11519-1 sind die physikalische Schicht sowie die Sicherungsschicht des CAN-Protokolls spezifiziert. Die Übertragungsrate kann demnach zwischen 1MBit/s bei 40m Buslänge und 50kBit/s bei 1000m Buslänge variieren. Die Signalübertragung erfolgt über eine Zweidrahtleitung

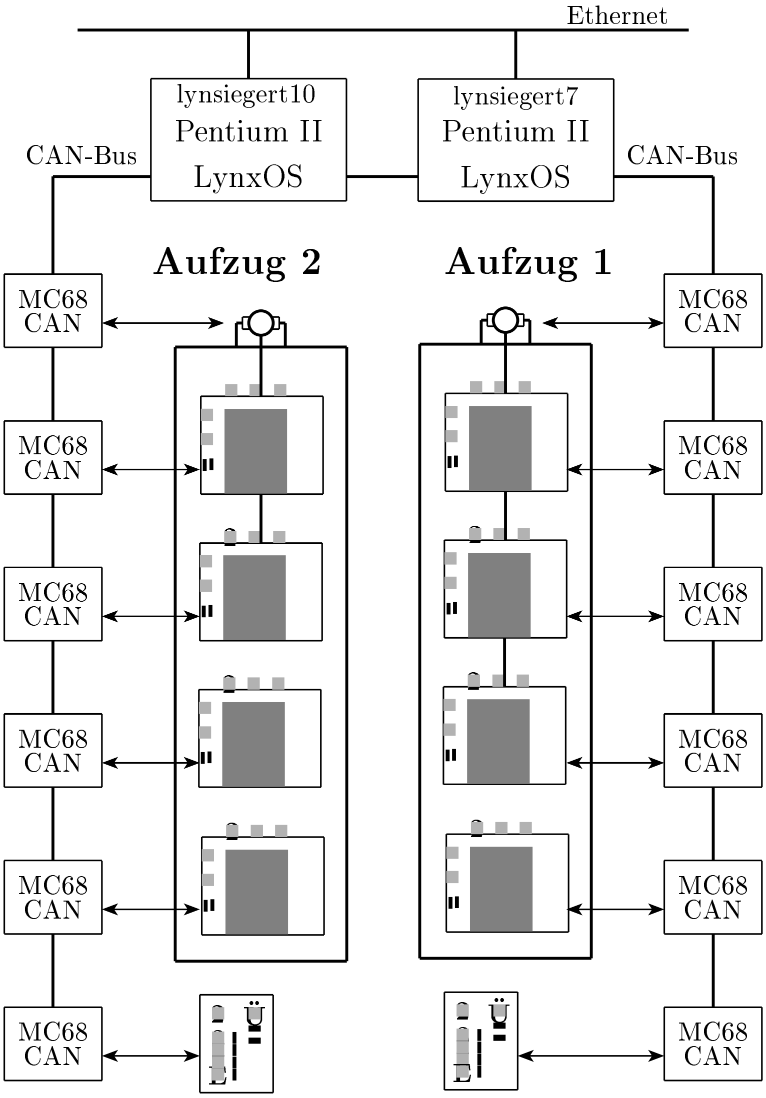


Abbildung 1: CAN-Bus-Struktur für Doppelaufzug

nach RS-485 bzw. ISO 11898.

Über den CAN-Bus werden Objekte mit einer genau definierten und normierten Objektstruktur übertragen. Ein solches Objekt besitzt zwei Byte mit Statusinformationen und bis zu acht Byte Nutzinformation (vgl. Abbildung 2). Ein CAN-Objekt kann also maximal 10 Byte groß sein. Die ersten 11 Bit der Statusinformation bilden den Objekt-Identifikator. Demnach sind 2048 verschiedene *Identifikatoren* möglich. Das 12. Bit bestimmt, ob es sich um einen *Transmit* oder einen *Remote Transmit Request* handelt und die verbleibenden 4 Bit der Statusinformation ergeben die Länge der Nutzdaten.

Beim Versand von Botschaften werden keine Empfangs- und Sendestationen adressiert. Jede Station kann jede über den CAN-Bus laufende Botschaft empfangen. Der Absender einer Botschaft ist unbekannt. Lediglich der Identifikator des Objekts auf dem Bus ist relevant. Der CAN-Bus ist somit rein objektorientiert.

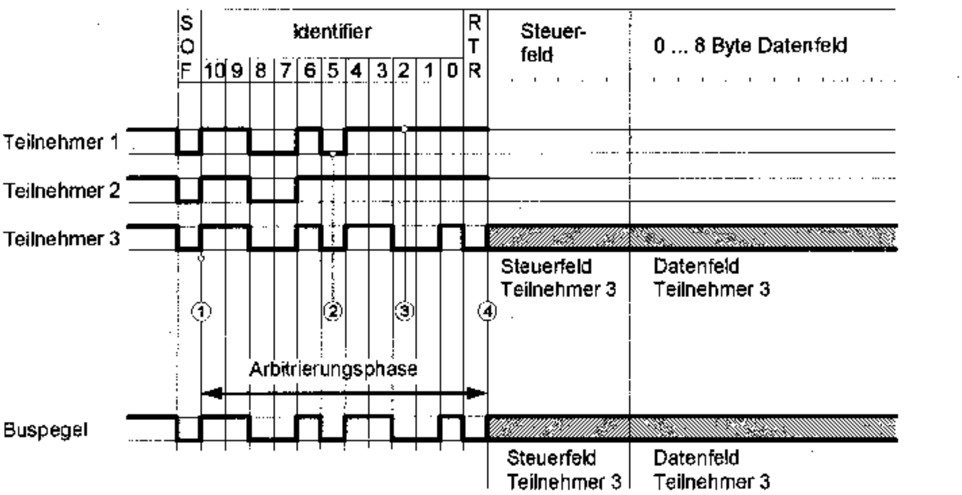
Auf der CAN-Bus-Leitung wird zwischen dominanten und rezessiven Bits unterschieden.

	Bits 7-0							
Byte 0	ID 10	ID 9	ID 8	ID 7	ID 6	ID 5	ID 4	ID 3
Byte 1	ID 2	ID 1	ID 0	RTR Bit	DLC 3	DLC 2	DLC 1	DLC 0
Byte 2-9	Objekt-Daten							

DLC: Data Length Coding
RTR: Remote Transmit Request

Abbildung 2: Aufbau eines CAN-Objekts

Ein nicht gesetztes Bit wird als dominanter Pegel übertragen und ein gesetztes Bit als rezessiver Pegel. Bei Zugriffskonflikten auf dem Bus können rezessive Pegel durch dominante überlagert werden, umgekehrt jedoch nicht.
Nach der Definition des CAN-Protokolls besitzt jede Botschaft einen eindeutigen Identifikator. Dieser Identifikator stellt beim Buszugriff nicht nur den Namen der Botschaft dar, sondern auch die Priorität der Botschaft.



Beispiel eines Arbitrierungsvorgangs im CAN-Protokoll. Die Teilnehmer 1, 2 und 3 beginnen gleichzeitig einen Arbitrierungsversuch (1). Teilnehmer 2 verliert zum Zeitpunkt (2), Teilnehmer 1 zum Zeitpunkt (3) das Buszugriffsrecht. Beide Teilnehmer gehen damit in den Empfangszustand; am Ende der Arbitrierungsphase (4) besitzt nur noch Teilnehmer 3 das Buszugriffsrecht und schaltet seine Nachricht auf den Bus.

Abbildung 3: Arbitrierung auf dem CAN-Bus

Falls zwei oder mehrere CAN-Controller gleichzeitig versuchen, ein Objekt über den Bus zu senden, wird der Konflikt durch bitweise Arbitrierung gelöst (siehe Abbildung 3). Die Sendelogik jedes Transmitters vergleicht die übertragenen Bitpegel mit den Bitpegeln auf dem Bus. Ist ein rezessives Bit auf dem Bus in ein dominantes Bit umgeschlagen, so bedeutet dies, daß gerade ein anderes Objekt mit höherer Priorität auf dem Bus übertragen werden soll. Die Sendelogik unterbricht hier den eigenen Übertragungsvorgang, bis das Objekt mit höherer Priorität gesendet wurde. Der Identifikator mit dem niedrigsten Binärwert hat so die höchste Priorität.

Aber Achtung: Die Arbitrierung erstreckt sich nur über die ersten 12 Bits. Daß heißt, sollten zwei Stationen Objekte mit gleichem Identifikator, aber unterschiedlichem Inhalt senden, kann dies zu Fehlern führen, die nicht aufgelöst werden können. Es wird ein Fehlerprotokoll initiiert, das die eben erfolgte Übertragung für ungültig erklärt. Beide Stationen versuchen nun anschließend ihre Nachricht zu wiederholen, was folglich wieder scheitert. Es liegt daher in der Verantwortung der Anwender solche Kollisionen zu vermeiden. (Z.B. keine zwei *Sende*stationen dürfen Objekte mit gleichen Identifikatoren besitzen.)

5.3 Implementierungshinweise

5.3.1 Zuteilung der Objektidentifikatoren zur Steuerung der Aufzugfunktionen

Die Nachrichten sprechen die Register der CAN-Platinen über eindeutige Identifikatoren an. Jede Nachricht enthält ein Byte Information, den Wert des entsprechenden Registers. Die Zuordnung der Identifikatoren erfolgt nach folgendem Schema aus drei Hex-Ziffern <*aufzug*><*platine*><*register*>. Um z.B. einen Wert in das Register für die Anzeigen im Stockwerk 2 des 1. Aufzugs zu verschicken, verwendet man den Identifikator 16#132#.

1. **Hex-Ziffer <*aufzug*>: Unterscheidung der Aufzüge**

Aufzug 1	<*aufzug*>=1
Aufzug 2	<*aufzug*>=2
Aufzug 3	<*aufzug*>=3

2. **Hex-Ziffer <*platine*>: Bezeichner für die Platinen der Aufzüge**

Kabinenmotor	<*platine*>=0
Erdgeschoss	<*platine*>=1
1. Stock	<*platine*>=2
2. Stock	<*platine*>=3
3. Stock	<*platine*>=4
Tastfeld	<*platine*>=7

3. **Hex-Ziffer $<register>$: Bezeichner für die E/A-Register der Platinen**

Taster	$<register>=0$
Motor	$<register>=1$
Anzeigen	$<register>=2$
LED-Anzeige	$<register>=3$

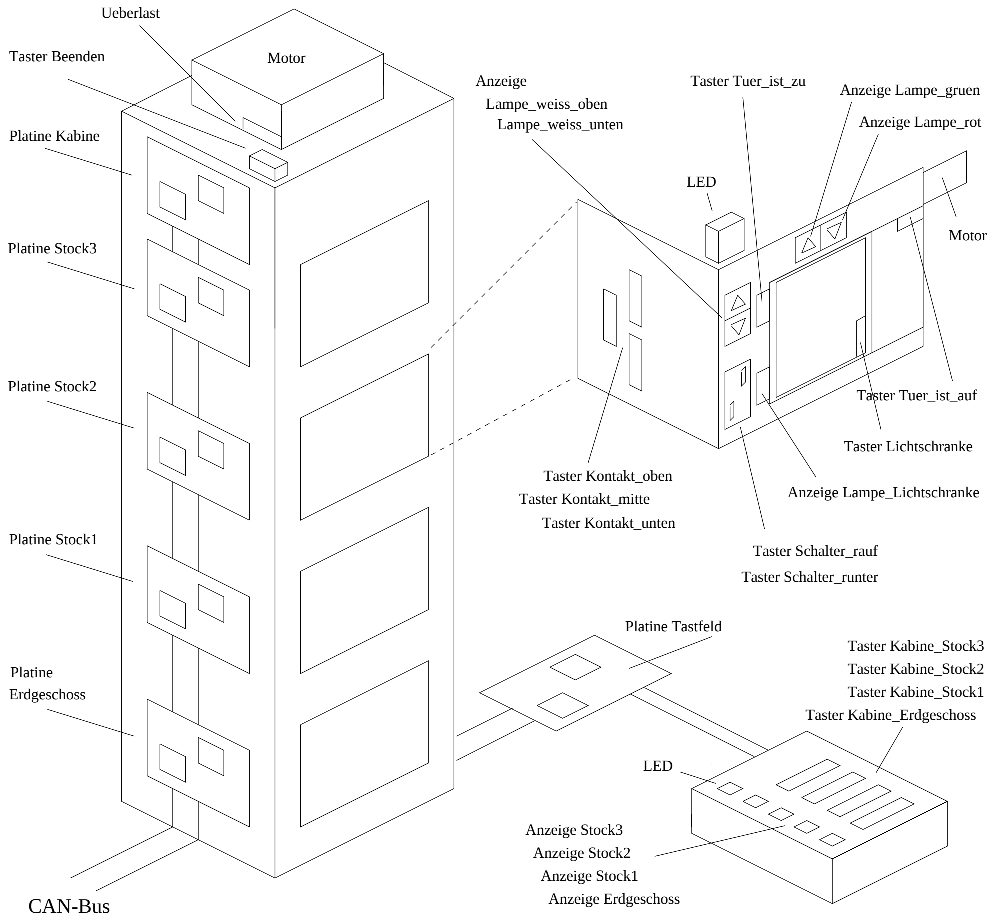


Abbildung 4: Schematische Darstellung des Modell-Aufzugs

5.3.2 Belegung der E/A-Register der CAN-Platinen

Zum Schalten der Motoren bzw. Lampen muß eine CAN-Nachricht mit dem entsprechenden Identifikator und einem Datenbyte gesendet werden; das Bitmuster des gesendeten Bytes steuert die einzelnen Motoren und Lämpchen. Jede Veränderung der Eingabewerte der Taster bzw. Lichtschranken bewirkt, daß eine Nachricht mit dem entsprechenden Identifikator über den CAN-Bus geschickt wird, die ein Datenbyte mit allen aktuellen Werten enthält. Dabei wird nicht nur das geänderte Bit gesendet, sondern das Abbild sämtlicher Eingabewerte der zugeordneten Platine. Es gilt folgende Zuordnung der E/A-Register zu den Aktoren und Sensoren des Aufzugs. In Abbildung 4 ist die Bedeutung der einzelnen Aktoren und Sensoren im Aufzugsmodell schematisch dargestellt. Für diese Aufgabe werden nicht alle aufgeführten Taster und Anzeigen gebraucht. Insbesondere haben Sie freie Wahl in den verwendeten Anzeigen (Lampen, LED).

- Platine für den Kabinenmotor:

Register	Identifikator	Hex-Wert	Aktor/Sensor
Taster	16#<*aufzug*>00#	1	Überlast
		4	Beenden
Motor	16#<*aufzug*>01#	1	Kabine rauf
		3	Kabine langsam rauf
		5	Kabine runter
		7	Kabine langsam runter

- Platinen für Stockwerke (<*platine*>=1,2,3,4):

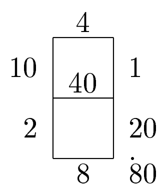
Register	Identifikator	Hex-Wert	Aktor/Sensor
Taster	16#<*aufzug*><*platine*>0#	1	Lichtschranke
		2	Tür ist auf
		4	Schalter rauf
		8	Schalter runter
		10	Tür ist zu
		20	Kontakt oben
		40	Kontakt mitte
		80	Kontakt unten
Motor	16#<*aufzug*><*platine*>1#	1	Tür auf
		5	Tür zu
Anzeigen	16#<*aufzug*><*platine*>2#	1	Lampe grün
		4	Lampe rot
		10	Lampe weiß oben
		40	Lampe weiß unten
		80	Lampe Lichtschranke

- Platine für Steuerung in der Kabine:

Register	Identifikator	Hex-Wert	Aktor/Sensor
Taster	16#<*aufzug*>70#	1	Fahrziel 3. Stock
		2	Fahrziel 2. Stock
		4	Fahrziel 1. Stock
		8	Fahrziel Erdgeschoß
		10	Alarm
		20	Schalter Nothalt
		40	Schalter Innensteuerung
Anzeigen	16#<*aufzug*>72#	1	3. Stock
		4	2. Stock
		8	weisse Lampe unten
		10	1. Stock
		20	Alarmhupe
		40	Erdgeschoß
		80	rote Lampe

- LED-Anzeige:

  Die LED Anzeige erhält in jedem Stockwerk und in der Kabine den einheitlichen Identifikator 16#<*aufzug*>73#.
  Bitzuordnug zu den Segmenten der Anzeige:



5.3.3 Der CAN-Bus Monitor

Mit dem Befehl `canmon /dev/canbus/canbus_qx` (wobei `x=1,2,3`) startet man den Monitor für den CAN-Bus, mit dem einzelne Steuerbotschaften verschickt und Meldungsbotschaften verfolgt werden können. Die Befehle sind selbsterklärend.

5.3.4 ADA-Package für CAN-Bus

Das ADAPackage `canada.ada` im Verzeichnis `/usr/local/LynxOS/ADA_LIBS/aufzug/` definiert die Schnittstellen zur CAN-Hardware.

Einbindung in ADA-Programm durch: "`with canada; use canada;`"
Beim Binden muß "`bind [name],MODULES=>canada.o`" angegeben werden. Hier eine Auswahl von Definitionen, Prozeduren und Funktionen, die Sie verwenden sollen:

```
package canada is

  type CANDATA is ARRAY(0..7) OF BYTE;

  type CANMODE is (MASTER,BUFFER,QUEUE);

  type CANOBJECT is
    record
      id    : INTEGER;     -- ObjektId
      state : INTEGER;     -- Statusinfo nur fuer receive !
      dlc   : BYTE;        -- Data Length Code
      data  : CANDATA;     -- Daten: 8 Bytes
    end record;

  function CANopen     ( mode  : in CANMODE;
                         modenr: in INTEGER;
                         ndelay: in BOOLEAN)     return INTEGER;

  function CANclose    ( fdes  : in INTEGER)     return BOOLEAN;

  function CANtransmit ( fdes  : in INTEGER;
                         object: in CANOBJECT)   return BOOLEAN;

  function CANconfig   ( fdes  : in INTEGER;
                         id    : in INTEGER)     return BOOLEAN;

  procedure CANreceive ( fdes  : in INTEGER; obj : out CANOBJECT);

end canada;
```

Die Funktion `CANopen` öffnet das CAN-Device mit der Nummer `modenr` in der Betriebsart `mode` und liefert als Rückgabewert einen gültigen Filedeskriptor. Auf den Rechnern des

Praktikums sind je drei exklusive logische Treiber (modenr = 1..3) für die Betriebsart BUFFER installiert. In dieser im Praktikum verwendeten Betriebsart werden alle Objekte in einem Puffer verwaltet. Wird ein neues Botschafts-Objekt empfangen, werden die Daten des entsprechenden Puffer-Objektes aktualisiert. Dies geschieht ohne Rücksicht darauf, ob die alten Daten vom Benutzer ausgelesen wurden. Setzen Sie den Parmeter `ndelay = FALSE`, um auf ankommende Nachrichten zu warten.

Mit der Funktion `CANconfig` wird ein Objekt mit dem Identifikator `id` für den Empfang konfiguriert. In der Buffer-Betriebsart werden nur konfigurierte Objekte empfangen. Der Benutzer empfängt also nur die für ihn relevanten Botschaften.

Die Funktionen `CANconfig` *und* `CANopen` *werden in dieser Aufgabe nicht direkt verwendet, sondern durch die Funktion* `CANinit` *aus dem package Aufzugdaten (siehe Kap. 5.3.5) implementiert.*

Die Funktion `CANtransmit` verschickt das CAN-Objekt `object` über das CAN-Device `fdes`. `fdes` muß mit dem durch die Funktion `CANopen` zurückgegebenen Filedeskriptor belegt werden. Der Rückgabewert ist `TRUE` bei Erfolg, ansonsten `FALSE`.

Mit der Prozedur `CANreceive` kann über das CAN-Device `fdes` eine Botschaft empfangen werden, wobei die Daten des Objekts in die Variable `object` eingetragen werden. Falls beim Empfang einer Botschaft ein Fehler aufgetreten ist, so wird `object.id` auf `-1` gesetzt.

Die Funktion `CANclose` schließt das CAN-Device `fdes`. Der Rückgabewert ist `TRUE` bei Erfolg, ansonsten `FALSE`.

5.3.5 ADA-Package Aufzugdaten

Das Paket `aufzugdaten` in der Datei `/usr/local/LynxOS/ADA_LIBS/aufzug/aufzugdaten.ada` definiert Datenstrukturen und eine Initialisierungsprozedur zum Zugriff auf die Register der Aufzugsteuerung.

Einbindung in ADA-Programm durch: “`with aufzugdaten; use aufzugdaten;`“

```
PACKAGE Aufzugdaten IS

--------------------------------------------
---------- Platinen und deren Register -----
--------------------------------------------
  SUBTYPE T_AufzugNr IS INTEGER RANGE 1 .. 3;

  TYPE T_Platine  IS (Kabine, Erdgeschoss, Stock1, Stock2, Stock3, Tastfeld);

  TYPE T_Register IS (Taster, Motor, Anzeigen, LED);
--------------------------------------------
------ Bits der Register -------------------
--------------------------------------------
----------------------------------------------------------------
--- Bei der Richtung sind es nicht die einzelnen Bits, sondern
--- komplette Steuerungsbefehle mit den jeweiligen BitMasken.
--- Dies macht die Programmierung etwas einfacher.
---------------------------------------------------------------
  TYPE T_Richtung IS (Kabine_rauf,Tuer_auf,Kabine_runter,Tuer_zu,
```

```
                      langsam_rauf,langsam_runter,anhalten,stop);

  TYPE T_Stock_Anzeigen IS (Lampe_gruen,Lampe_rot,Lampe_weiss_oben,
                            Lampe_weiss_unten,Lampe_Lichtschranke);

  TYPE T_Kabine_Taster IS (Ueberlast,Beenden);

  TYPE T_Stock_Taster IS (Lichtschranke,Tuer_ist_auf,Schalter_rauf,
                          Schalter_runter,Tuer_ist_zu,Kontakt_oben,
                          Kontakt_mitte,Kontakt_unten);

  TYPE T_Tastfeld_Taster IS (Kabine_Alarm,Kabine_Erdgeschoss,Kabine_Stock1,
                             Kabine_Stock2,Kabine_Stock3,Kabine_Nothalt,
                             Kabine_Innensteuerung);
-------------------------------------------------------
----- Identifikatoren der Platinen und Register -------
-------------------------------------------------------
  AufzugId   : ARRAY (T_AufzugNr) of INTEGER :=
               (16#100#,16#200#,16#300#);
  PlatineId  : ARRAY (T_Platine)  of INTEGER :=
               (16#000#,16#010#,16#020#,16#030#,16#040#,16#070#);

  RegisterId : ARRAY (T_Register) of INTEGER := (0,1,2,3);
-------------------------------------------------------
----- Werte der einzelnen Bits ------------------------
-------------------------------------------------------
  MBits : ARRAY (T_Richtung) of BYTE    :=( Kabine_rauf  =>16#01#,
                                            Tuer_auf     =>16#01#,
                                            Kabine_runter=>16#05#,
                                            Tuer_zu      =>16#05#,
                                            stop         =>16#0#,
                                            anhalten     =>16#0#,
                                            langsam_rauf =>16#03#,
                                            langsam_runter=>16#07# );

  SABits : ARRAY (T_Stock_Anzeigen) of BYTE:=(Lampe_gruen        =>16#01#,
                                              Lampe_rot          =>16#04#,
                                              Lampe_weiss_oben   =>16#10#,
                                              Lampe_weiss_unten  =>16#40#,
                                              Lampe_Lichtschranke=>16#80#);

  KBits : ARRAY (T_Kabine_Taster) of BYTE:=(Ueberlast=>16#01#,
                                            Beenden  =>16#04#);

  SBits : ARRAY (T_Stock_Taster) of BYTE:= (Lichtschranke  =>16#01#,
                                            Tuer_ist_auf   =>16#02#,
                                            Schalter_rauf  =>16#04#,
                                            Schalter_runter=>16#08#,
                                            Tuer_ist_zu    =>16#10#,
                                            Kontakt_oben   =>16#20#,
                                            Kontakt_mitte  =>16#40#,
                                            Kontakt_unten  =>16#80#);

  TBits : ARRAY (T_Tastfeld_Taster) of BYTE:= (Kabine_Stock3 =>16#01#,
                                               Kabine_Stock2 =>16#02#,
                                               Kabine_Stock1 =>16#04#,
                                               Kabine_Erdgeschoss =>16#08#,
                                               Kabine_Alarm  =>16#10#,
                                               Kabine_Nothalt=>16#20#,
                                               Kabine_Innensteuerung=>16#40#);

-------------------------------------------------------------------
-- Bits fuer die weissen Laempchen in der Kabine und die Hupe
-- geordnet nach Stockwerken(Typ Platine), die aufleuchten koennen.
-- Das Ueberlast-Bit wird einfach gleich dem Bit fuer das Bit 'Kabine'
-- gesetzt, wodurch mit KABits(Kabine) das rote Laempchen
-- referenziert werden kann. Ebenso wird mit 'Tastfeld' die Hupe
-- referenziert. Dies spart die Deklaration eines eigenen Typs
-------------------------------------------------------------------
  KABits : ARRAY (T_Platine)  OF BYTE := (Erdgeschoss =>16#40#,
```

```
                                          Stock1      =>16#10#,
                                          Stock2      =>16#04#,
                                          Stock3      =>16#01#,
                                          Kabine      =>16#80#, -- Ueberlast
                                          Tastfeld    =>16#20#);-- Hupe
-------------------------------------------
-- Die LED-Werte fuer die Stockwerksanzeige
-------------------------------------------
  LBits : ARRAY (T_Platine) of BYTE := (16#0#,16#BF#,16#A1#,
                                      16#CF#,16#ED#,16#0#);
-----------------------------------------------------
----- sonstige globale Variablen --------------------
-----------------------------------------------------

  fdes      : INTEGER;           -- Filehandle fuer Zugriff auf den CAN-Bus
  Akzeptanz : ACCEPTANCE;        -- Akzeptanzmaske fuer Aufzugsteuerung

  FUNCTION caninit (Aufzug: INTEGER; Buffer_nr: INTEGER) RETURN INTEGER;
END Aufzugdaten;
```

Mit der Funktion `CANinit` wird der CAN-Bus Zugriff auf Aufzug 1,2 oder 3 initialisiert. Es wird das Device im Buffer-Modus geöffnet, die Akzeptanzmaske richtig besetzt und die erforderlichen Empfangspuffer werden eingerichtet. Beim Aufruf von `CANinit` muß der formale Parameter `Aufzug` mit der Aufzugnummer (1,2,3) und `Buffer_nr` mit der Nummer eines Buffer-Devices (1,2,3) belegt werden. Es ist sinnvoll, diese beiden Parameter erst zur Laufzeit einzulesen, da dann flexibel auf den gerade verfügbaren Aufzug bzw. auf das freie Buffer-Device zugegriffen werden kann. Der Rückgabewert ist entweder ein gültiger Filedeskriptor oder -1 bei einem Fehler. `CANinit` ruft `CANopen` und `CANconfig` mit passenden Parametern auf und initialisiert Variable im Paket Aufzugdaten.

5.3.6 Weitere Hinweise

- Man sollte nach Möglichleit die in dem Paket `aufzugdaten` gemachten Definitionen verwenden.
  z.B. `MBits(Kabine_runter)` statt `16#05#`, das erhöht die Lesbarkeit wesentlich.
- Möchte man auf die `BYTE`-Werte bool'sche Verknüpfungen anwenden, so muß man zuvor das Standart-Paket `UNSIGNED` mittels einer `WITH`- und USE-Klausel sichtbarmachen. In diesen Paket werden nicht nur die Typen `BYTE` und `WORD` definiert, sondern auch die darauf anwendbaren Funktionen `AND`, `OR`, `NOT` und `XOR`. Damit können Sie einfach auf die einzelnen Bits (Taster, Lampen, Motoren) der Register zugreifen:
  ```
  o: CANOBJECT;
  if ( o.data(0) AND KBits(Beenden) ) /= 0 THEN -- Endetaster gedrückt
  ```
- Es ist ratsam, eine Initialisierungs-Prozedur zu schreiben, die den Aufzug bei Programmstart in einen definierten Anfangszustand überführt.
- Um Ihr Programm korrekt zu verlassen, verwenden Sie bitte den Beenden-Taster oben-links am Aufzugmodell. Wenn Ihr Programm die Betätigung dieses Tasters feststellt, sollten sie sicherstellen das keiner Ihrer Aktoren – vor allem Motoren – noch in Betrieb ist. Sie sollten diese abschalten, bevor Sie Ihr Programm endgültig verlassen. Vergessen Sie nicht, das CAN-Device mit CANclose zu schließen.

6 Vier gewinnt in Echtzeit

6.1 Aufgabenstellung

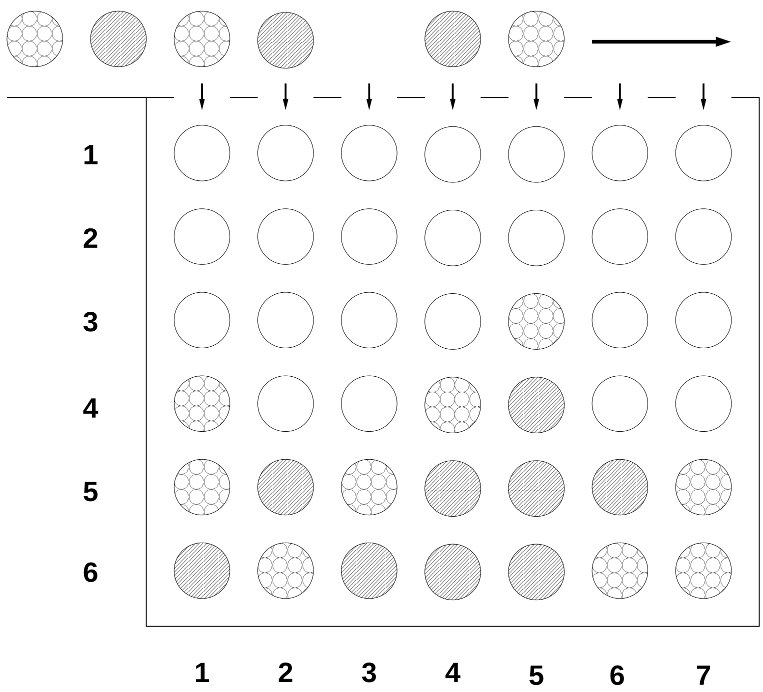


Abbildung 5: Das Spielfeld

6.1.1 Das Spiel

Das Spielfeld von Abbildung 5 und folgender Spielablauf werden auf einem Rechner (Spielserver) simuliert:
Insgesamt 21 rote und 21 blaue Chips werden alternierend in der Einwurfzeile von links nach rechts im vorgegeben Takt über das Spielfeld geschoben. Die Spieler können in jeder Runde beliebig viele Chips der eigenen Farbe ins Spielfeld fallen lassen. Nicht verbrauchte Chips fallen nach rechts heraus. Gewonnen hat, wer als erster *vier* Chips in seiner Farbe in einer Reihe (waagrecht, senkrecht, diagonal) hat.

Informationen vom Spielserver und Steuerungsbefehle der Spieler werden als Nachrichten über den CAN-Bus geschickt. In der Testphase stehen vier Spielserver bereit, so dass jede Gruppe unabhängig testen kann. Das Turnier wird mit dem Server 1 durchgeführt. Die Nachrichten haben folgenden Aufbau (Notation siehe Typdefinitionen in Kapitel 5.3.4). Auf die Identifikatoren für Server 2,3,4 ist jeweils 16#20#,16#40# bzw. 16#60# zu addieren.

- Spielstart (vom Spielserver 1)

canobjekt	Information für		
	Spieler 1	Spieler 2	
id	16#21#	16#22#	
byte(0)	Δ t	Δ t	Taktzeit in 1/10 sek (1..50)
byte(1)	1/2	1/2	zugeteilte Farbe im Spiel 1/2 für rot/blau rot beginnt

- Nach jeder Runde (vom Spielserver 1)

canobjekt	Information	
id	16#10#	
byte(0)	Runde (1..49)	Abgelaufenen Runde
byte(1)	0/1/2	Spalte 1..7 der
byte(2)	0/1/2	Einwurfzeile am Ende dieser Runde
⋮	⋮	0/1/2 für leer/rot/blau
byte(7)	0/1/2	

- Am Ende des Spiels (vom Spielserver 1)

canobjekt	Information	
id	16#20#	
byte(0)	Runde (1..49)	Siegesrunde
byte(1)	1/2/3	Sieg rot / Sieg blau / unentschieden

- Nachricht der Spieler an den Server 1 zum Einwurf eines Chips

canobjekt	Information	
id	16#10# + s	s Index der Einwurfspalte (1..7)
byte(0)	1/2	1/2 für rot/blau

Ist ein eigener Chip dieser Farbe an dieser Spalte der Einwurfzeile, so fällt er ins Spielfeld auf den obersten nicht besetzten Platz. Ist die Spalte bereits aufgefüllt, so bleibt er in der Einwurfzeile. Falls kein Chip oder kein Chip der eigenen Farbe an dieser Stelle ist, wird keine Operation ausgeführt.

Nach *jedem* Einwurf wird berechnet, ob das Siegkriterium erfüllt ist und gegebenenfalls das Spiel mit der Siegnachricht beendet.

6.1.2 Die Aufgabe

1. Schreiben Sie zunächst ein Ada-Programm, welches die Ausgaben des Spielservers als Abbild des Spielfeld protokolliert und ein manuelles Spiel erlaubt.
2. Entwickeln Sie eine Strategie für das Spiel *Vier Gewinnt* mit Zeitbedingungen. Da Sie nicht beliebig lange rechnen können, müssen Sie geeignete Heuristiken einsetzen.
3. Implementieren Sie Ihre Strategie in einem Ada-Programm. Zeitbedingung können Sie über timeout in der select-Anweisung realisieren.
4. Geben Sie das kommentierte Programmlisting und eine Dokumentation Ihrer Strategie im DIN A4-Format geheftet ab.
5. Am Ende des Praktikums wird in einem Turnierspiel entschieden, wer das beste Spielprogramm implementiert hat. Dazu sollen alle Gruppen gegeneinander je zwei Spiele austragen. Folgende Punktwertung gilt:

Verlust:	0 Punkte
Unentschieden:	1 Punkt
Sieg:	3 Punkte

Die Gruppe mit der höchsten Punktezahl hat gewonnen.

6.2 Implementierungshinweise

6.2.1 ADA-Schnittstellen-Pakete

Die Initialisierung des CAN-Bus ist mit `CANopen` und `CANconfig` selbst durchzuführen; es gibt keine vorgefertigte Initialisierungsprozedur. Weitere Informationen in Kap. 5.3.4

6.2.2 Der Spielserver

Laden Sie für den Spielserver von `/usr/local/LynxOS/ADA_LIBS/viergewinnt/` folgende Programme:
`VierServer, VierGewinnt.tcl, VierGewinnt`.
Starten Sie den Spielserver nach Ihrem Programm mit `VierGewinnt`. Der Server erlaubt auch manuelle Eingabe und einen Computergegner als Spieler 2.

6.2.3 Weitere Hinweise

- Ein Rahmenprogramm zur Lösung der Aufgabe steht unter /usr/local/LynxOS/ADA_LIBS/viergewinnt/spielfeld.ada zum Download bereit. Damit können Sie mehr Zeit auf die Spielstrategie verwenden!
- Um die Aufgabe nicht unnötig zu komplizieren, wurden keine Sicherungsmaßnahmen eingebaut, die ein Auslösen der Chips des Gegners verhindern. Bitte versuchen Sie keine faulen Tricks!